

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

GÉNÉRATEUR DE PHRASES BASÉ SUR UNE ONTOLOGIE SYNTAXIQUE

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN LINGUISTIQUE

PAR
KARL SZYMONIAK

MAI 2011

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.01-2006). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

Je tiens en premier lieu à remercier Denis Foucambert, mon directeur, ainsi que Roger Nkambou, mon codirecteur, d'avoir accepté de me suivre et de me soutenir lorsque je me suis jeté dans ce projet de générateur de phrases.

Je veux également remercier mes lectrices, Louisette Emirkanian et Sophie Piron, pour leur soutien et leur conseil dans les moments de doute. Un merci tout particulier à Sophie qui m'a permis de revenir travailler à l'UQÀM et sans qui je ne serais pas là.

Un grand merci à tous mes collègues du Groupe de recherche sur la LSQ et le bilinguisme sourd pour m'avoir supporté pendant l'écriture de ce mémoire : Anne-Marie Parisot, Dominique Machabée, Amélie Voghel, Julie Rinfret, Caroline Hould, Marc-André Bernier, sans oublier Lynda Lelièvre. Merci d'avoir supporté mes déprimés et mes excès durant ces dernières semaines.

Un merci particulier à mes collègues étudiants et plus particulièrement à Geneviève Domingue pour nos longues discussions et ses conseils avisés.

Merci à tous mes amis de France et d'ici pour leur soutien : Gaëlle et Fred, Stéphanie, Aline et Sylvain, Marilys et Rémy, Caroline et François, Typhaine et Julien, Nathalie M., Stéphanie, Catherine, Martin, Anne, Pier-Luc, Nathalie S., Norman et Walter, Mélodie, Norman L., Denis, Claude, Bruno, Julie, Yves et Laurent, Elsa et Batiste et les autres.

Avant de finir, un merci à ma famille présente malgré la distance (merci Skype!) et qui me soutient depuis toujours dans mes choix les plus hasardeux comme dans ceux plus réfléchis tel que ma reprise d'étude au Québec.

Pour finir, je voudrais remercier et dédier ce mémoire à mon mari, Vincent, pour sa présence, son aide et sans qui rien de tout cela n'aurait été possible.

TABLE DES MATIÈRES

LISTE DES FIGURES.....	vii
LISTE DES ACRONYMES	ix
RÉSUMÉ	x
INTRODUCTION	1
CHAPITRE I	
PROBLÉMATIQUE ET CADRE THÉORIQUE	5
1.1 Historique.....	6
1.1.1 La période «Classique»	6
1.1.2 Le XXe siècle	11
1.2 Génération.....	20
1.2.1 Les questions fondamentales.....	22
1.2.2 La génération profonde	24
1.2.3 La génération de surface	32
1.2.4 Conclusion	36
1.3 Logiciels	36
1.3.1 Tree-Tagger	37

1.3.2	L'ontologie par Protégé.....	39
1.3.3	LKB	44
1.5	Conclusion	46

CHAPITRE II

MÉTHODOLOGIE.....	47
2.1 Corpus	49
2.2 Langage	51
2.3 Sous-programmes	52
2.3.1 Programme de récupération des données dans le corpus	53
2.3.2 Programme de création de phrases.....	55
2.3.3 Programme de mise en mots	57
2.3.4 Programme d'étiquetage	58
2.3.5 Programme de « re-création » des phrases	59
2.3.6 Programme de découpage en syntagmes.....	60
2.3.7 Conclusion.....	63
2.4 Réalisation de l'ontologie	64
2.5 Réalisation du programme de génération	68
2.6 Conclusion	71

CHAPITRE III

RÉSULTATS	72
3.1 Problèmes rencontrés et solutions alternatives.	72
3.1.1 Les sous-programmes.....	73
3.1.2 Le LKB	75
3.1.3 L'ontologie	76

3.1.4	Le programme	77
3.2	Résultats	79
3.2.1	Les sous-programmes	79
3.2.2	L'ontologie	80
3.2.3	Le générateur	83
3.3	Conclusion	86
CONCLUSION		87
ANNEXES A		
LES SOUS-PROGRAMMES		89
1	01-Projet_memoire_recup.pl.....	89
2	02-Projet_memoire_phrases.pl.....	97
3	03-Projet_memoire_syntagm_tag.....	102
ANNEXE B		
PROGRAMME DE GÉNÉRATION.....		112
1	Generateur.pl	112
2	module1.pl.....	117
ANNEXE C		
EXEMPLE DE RÉSULTATS		123
1	Génération de la structure de base.....	123
2	Génération de phrase.	130

ANNEXE D	
CATÉGORIES DU TREE-TAGGER	135
 BIBLIOGRAPHIE	 136

LISTE DES FIGURES

Figure 1 :	Résonateurs de Kratenstein.	7
Figure 2 :	Image d'une machine à parler du baron Von Kempelen.	9
Figure 3 :	Fonctionnement de la machine à parler du baron Von Kempelen d'après la description de Wheatstone (1802-1875) au XIXe siècle.	9
Figure 4 :	Arbres élémentaires pour la phrase : 'Jean dort'.	26
Figure 5 :	Structure simplifiée de la phrase 'Jean dort'.	28
Figure 6 :	Structure du SN <i>un livre</i>	30
Figure 7 :	Exemple de graphe utilisé dans la génération.	35
Figure 8 :	Étiquetage des syntagmes de la phrase «Il la porte la porte.»	37
Figure 9 :	Étiquetage des syntagmes de : «Il ferme d'une main ferme la porte de la ferme».	38
Figure 10 :	Exemples d'individus présents dans une ontologie.	41
Figure 11 :	Exemples de propriétés entre les individus d'une ontologie.	41
Figure 12 :	Cas d'individus identiques, mais ayant un nom différent.	42
Figure 13 :	Propriété transitive.	43
Figure 14 :	Propriété symétrique et asymétrique.	43
Figure 15 :	Propriété réflexive et non réflexive.	43
Figure 16 :	Exemple de classes.	44
Figure 17 :	Exemple de grammaire à produire pour la réalisation d'une structure de traits	45
Figure 18 :	Organigramme de la méthodologie.	48

Figure 19 :	Nouvel organigramme de la méthodologie.	48
Figure 20 :	Arborescence du répertoire « corpus ».	50
Figure 21 :	En-tête et premières lignes d'un fichier XML.	50
Figure 22 :	En-tête et premières lignes d'un fichier texte.	51
Figure 23 :	Boîte de dialogue initiale	53
Figure 24 :	Boîte de dialogue secondaire.	54
Figure 25 :	Extrait d'un fichier de phrases.	56
Figure 26 :	Extrait d'un fichier de mots.	57
Figure 27 :	mots taggés.	58
Figure 28 :	phrases taggées.	60
Figure 29 :	Phrases de lemmes.	60
Figure 30 :	Étiquetage des syntagmes de la phrase «Il ferme d'une main ferme la porte de la ferme».	62
Figure 31 :	Découpage d'un syntagme par ligne.	63
Figure 32 :	Premier niveau de l'ontologie.	65
Figure 33 :	Propriété liant le déterminant au nom.	66
Figure 34 :	Hiérarchisation des classes à l'aide du logiciel.	67
Figure 35 :	Hiérarchie de mes classes après et avant l'utilisation du raisonneur.	67
Figure 36 :	Schéma de mon algorithme.	68
Figure 37 :	Boîtes de dialogue de générateur.	69
Figure 38 :	Explication sur l'utilisation de la classe 'VERB_INTRANSITIF'.	70
Figure 39 :	En-tête et premières lignes d'un fichier texte du corpus.	73
Figure 40 :	URI.	78
Figure 41 :	Syntagmes constituant une des phrases de mon corpus après traitement.	79
Figure 42 :	Inférences réalisées sur les classes de mon ontologie.	81
Figure 43 :	Schéma de mes classes implémentées.	82
Figure 44 :	Schéma de mes classes inférées.	82
Figure 45 :	Partie d'une structure de phrase générée.	83
Figure 46 :	Résultat de deux générations par mon programme.	85

LISTE DES ACRONYMES

A.L.P.A.C.	Automatic Language Processing Advisory Committee
A.T.A.L.A.	Association pour l'étude et le développement de la Traduction Automatique et de la Linguistique Appliquée
A.T.N.	Réseaux de Transitions Augmentés
CPAN	Comprehensive Perl Archive Network
G.A.T.	Génération Automatique de texte
H.P.S.G.	Head-driven Phrase Structure Grammar
I.A.	Intelligence Artificielle
I.C.L.U.S	Institute for Computational Linguistics of University of Stuttgart
O.W.L.	Web Ontology Language
R.D.F.	Resource Description Framework
T.A.	Traduction Automatique
T.A.G.	Tree Adjoining Grammar
T.A.L.	Traitement Automatique de la Langue
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

RÉSUMÉ

Loin des automates du début du XVII^e siècle, le traitement automatique de la langue connaît dernièrement une évolution rapide notamment dans le domaine de l'analyse de corpus. Dans le cadre de la génération, les travaux restent cependant plus rares, principalement en raison de la complexité de leur implémentation. En plus des difficultés propres à la création de phrases ou de textes, un tel outil doit être en mesure d'effectuer une analyse du thème de l'interaction afin d'y conformer sa production.

L'objectif de mon mémoire est la génération de phrases aléatoires, donc sans analyse de thème, syntaxiquement et sémantiquement correctes.

Pour le côté syntaxique, j'ai décidé de construire mon programme autour d'une ontologie syntaxique basée sur un corpus du journal « *le Monde* ». Le but de cette ontologie syntaxique est de permettre la génération de phrases ayant une structure syntaxique correcte.

Le côté sémantique devait s'effectuer par l'ajout à cette structure d'un lexique marqué et permettre la génération de phrases sémantiquement correctes. Ce lexique devait être marqué à l'aide des structures de traits des grammaires syntagmatiques guidées par les têtes. Malheureusement, certains problèmes logistiques m'ont obligé à utiliser un vocabulaire marqué catégoriellement et, par là même, à abandonner la partie sémantique de la génération.

Mon projet de mémoire traite donc de la création d'un générateur de phrases, de sa conception à sa réalisation.

INTRODUCTION

Dans le cadre de mon projet, j'ai choisi de créer un générateur de phrases basé sur une ontologie syntaxique.

À l'heure actuelle, la génération est utilisée dans différents domaines aussi divers qui vont, par exemple, du médical¹ au météorologique². En effet, en fournissant les informations nécessaires, des générateurs de textes écrivent les recommandations pour certains patients ou des bulletins météorologiques destinés à la publication dans certains journaux, notamment en Angleterre ou au Canada.

De la même façon, les transports ferroviaires allemands utilisent la génération lorsqu'un client appelle le service pour connaître l'horaire d'un train. Les phrases produites sont des phrases à trous qui seront remplies en fonction du résultat de l'analyse de la demande du client avant d'être prononcées par un générateur de synthèse vocale.

Par ailleurs, du côté de la recherche, les linguistes utilisaient la génération dans le but de vérifier la validité de théories linguistiques.

¹ Le système STOP, utilisé par NHS (British Health Service) et qui fournit à chaque patient un programme personnalisé pour arrêter de fumer.

² Le système FoG, mis en place par Environnement Canada et qui à partir d'une représentation graphique ou numérique génère un texte informatif.

Toutefois, la relation Homme-Machine ne peut se faire sans un dialogue compréhensible par les deux parties. La génération permet cette communication qui se développe de plus en plus, surtout dans le cadre des systèmes tutoriels intelligents (STI). C'est dans cette optique que la génération se développe depuis ces dernières années et c'est dans cette optique que je développerai mon générateur.

Ce projet de générateur de phrases a pour but de s'intégrer dans un programme un peu plus vaste que je souhaiterais élaborer et développer durant mon doctorat. Ce projet concerne la réalisation d'un système tutoriel intelligent dont le but sera d'enseigner aux professeurs de français la pédagogie et le vocabulaire adaptés aux élèves déficients auditifs dans le cadre de l'apprentissage de la lecture et de l'écriture du français. Mon générateur sera à la base de la génération des dialogues qui se feront entre le tuteur et l'enseignant.

L'intégration dans mon générateur d'une ontologie syntaxique va permettre lui apporter une connaissance syntaxique. En effet, une ontologie est la représentation d'un domaine de connaissance, d'un champ d'informations. Cette représentation est composée de termes ou de concepts, nommés classes, organisés en taxinomie autour de propriétés créant des liens hiérarchiques entre chaque classe (Gruber, 1993). Contrairement aux ontologies développées jusqu'à aujourd'hui, qui sont majoritairement axées sur la sémantique et le lexique, mon ontologie va être basée sur des informations syntaxiques.

Mais comment la représentation d'un domaine de connaissances peut-elle permettre, ou faciliter, la production d'un texte ou d'une phrase? En 1960, Yehoshua Bar-Hillel (Bar-Hillel, 1960) avait déjà trouvé la réponse à cette question. Pour ce dernier, qui travaillait dans le cadre de l'Intelligence Artificielle (I.A. dans le reste de ce mémoire), un programme doit avoir une connaissance du monde qui l'entoure afin de pouvoir comprendre les actions qu'il effectue. Toutefois, bien que Bar-Hillel ait eu une idée claire des besoins d'un programme, qu'il s'agisse de l'I.A. ou des programmes de traduction et de génération, le déploiement technologique à mettre en place pour réaliser ces représentations n'était tout simplement pas réalisable à son époque. Cependant, ce qui était alors techniquement impossible il y a cinquante ans n'est plus un défi et est réalisable sans avoir besoin de faire appel à des technologies complexes. Et l'un des moyens les plus efficaces pour représenter un domaine de connaissance est la représentation ontologique de ce domaine.

Depuis plusieurs années, un grand nombre d'ontologies ont été élaborées au sein de différents laboratoires. On retrouve des ontologies explicitant diverses théories de production de la langue, telles que les ontologies sens-textes (Tremblay, 2009) basées sur les théories d'Igor Mel'čuk, ou d'autres encore basées sur la sémantique telle que la SWEET Ontologies³. Hélas, malgré la diversité des ontologies créées, très peu sont basées sur la syntaxe.

Or, il me semble que la syntaxe est un élément essentiel dans la génération, au moins au même titre que la sémantique qui semble être beaucoup plus porteuse dans la recherche en génération. Pourtant, il existe à l'heure actuelle un grand nombre de recherches développant des grammaires formelles, lesquelles sont majoritairement tournées vers l'analyse. Cela est bien dommage, car certaines de ces grammaires, notamment les grammaires de traits telles que la Head-driven Phrase Structure Grammar (H.P.S.G. ou Grammaire Syntagmatique Guidée par les Têtes) développée par Carl Pollard et Ivan Sag dans les années 1980 (Pollard et Sag, 1994), sont tout à fait à même de permettre, voire de faciliter, la génération d'un texte et encore plus d'une phrase. D'ailleurs, depuis quelques années, Wilcock travaille sur la création d'une ontologie syntaxique basée sur ces grammaires (Wilcock, 2007).

Les grammaires H.P.S.G. permettent d'intégrer dans une structure de traits complexes un grand nombre d'informations. Ces informations, en plus d'être syntaxiques, peuvent être phonologiques, sémantiques, lexicales, morphologiques ou pragmatiques. Mais l'utilisateur peut également « créer », dans une certaine limite, des traits qui lui seront nécessaires dans l'élaboration de sa grammaire.

Le fait d'intégrer directement dans la structure de la phrase des informations sémantiques et lexicales peut permettre au générateur de contrôler la validité des productions tel que l'ordre des mots, l'accord entre les éléments lexicaux (par exemple : un nom masculin s'accorde avec un déterminant masculin), etc. Ainsi, la présence de ce type d'informations devrait permettre à la génération d'ajouter un aspect sémantique à l'aspect syntaxique de la production.

³ Une ontologie sémantique pour le Web sémantique ayant pour thème la Terre et la terminologie environnementale, développé par Jet Propulsion Laboratory.

Néanmoins, à mon niveau, l'implémentation d'une telle grammaire reste un réel défi que je voulais tenter d'aborder. En effet, j'avais l'intention d'utiliser la structure de trait de ces grammaires dans mon projet afin de compléter les structures fournies par mon ontologie avec un lexique marqué à la fois syntaxiquement et sémantiquement.

Pour ce faire, j'ai découpé ce mémoire en trois parties, avec pour commencer un chapitre sur l'historique. La première partie relate le développement du traitement automatique de la langue depuis le XVIIe siècle. La seconde partie portera sur l'histoire de la génération sous la forme d'état de l'art. Ce premier chapitre a donc pour but d'expliquer où en est la recherche à l'heure actuelle et donc de me permettre de poser clairement le cadre théorique dans lequel s'insère mon mémoire ainsi que d'exposer mes objectifs avec la création de ce générateur.

Dans le second chapitre, je vais décrire la méthodologie développée pour mettre au point mon générateur de façon cohérente. J'y exposerai l'acquisition de mon corpus, ainsi que les traitements réalisés, à travers les programmes développés, afin de faciliter son intégration dans l'ontologie. La présentation de la construction de l'ontologie sera expliquée dans une seconde partie de ce chapitre, suivie par des explications sur le programme de génération.

Dans un troisième chapitre, je reviendrai sur les difficultés rencontrées au cours de ce projet. Puis, au moyen de la présentation de quelques phrases issues de mon générateur, j'exposerai les réussites, les résultats et les problèmes inhérents à la génération.

Enfin, je passerai en revue certaines perspectives révélées par l'analyse de mon travail.

CHAPITRE I

PROBLÉMATIQUE ET CADRE THÉORIQUE

Le traitement automatique de la langue est un domaine principalement tourné vers l'analyse de la langue et dans lequel la génération peine à trouver sa place en tant qu'entité de recherche à part entière de la linguistique. Cela peut s'expliquer par le fait qu'elle est, depuis le début, l'apanage des informaticiens. Son appropriation par les linguistes est d'autant plus difficile qu'ils ne semblent s'y intéresser que pour vérifier leurs théories sur la production de la langue.

La création d'un générateur pourrait permettre de montrer que la génération de phrases ou de textes a une place concrète dans la linguistique en dehors de la vérification d'hypothèses langagières. Sans aller jusqu'à l'implémentation d'un générateur de textes, le but de mon projet est de mettre au point un générateur de phrases dont la production sera basée sur une ontologie syntaxique que je vais développer. Comme je vais l'expliquer dans la suite de ce chapitre, de nombreuses ontologies ont déjà été développées, mais aucune de ces ontologies n'est d'ordre syntaxique.

La place d'une ontologie syntaxique dans la génération de phrases, ou de textes, est pertinente en raison du fait qu'une ontologie permet de classer les éléments qui la composent de façon hiérarchique. Cette hiérarchie est basée sur les diverses propriétés qui unissent ces éléments. Dans le cas d'une ontologie syntaxique, les éléments d'un syntagme peuvent ainsi être définis par leurs liens avec le syntagme, par exemple : un syntagme nominal (SN) est composé d'un nom qui est, en général, la tête du syntagme. Mais le nom est, en principe,

précédé d'un déterminant. Ainsi, dans la construction de mon ontologie, le syntagme nominal sera composé d'un déterminant suivi d'un nom.

À l'aide de l'ontologie syntaxique, le générateur va créer une structure correcte de la phrase qu'il ne restera plus qu'à remplir avec un vocabulaire adapté. Le vocabulaire de trait fourni par la HPSG devait permettre de remplir l'ensemble de la structure en tenant compte de la sémantique et des informations lexicales.

Même si elle est depuis longtemps dans les mains des informaticiens, la génération a sa place en linguistique, car elle aborde des domaines appartenant à cette dernière tel que la syntaxe ou la sémantique.

Le bref historique suivant va permettre de comprendre pourquoi les générateurs sont généralement réalisés par des informaticiens.

1.1 Historique

1.1.1 La période «Classique⁴»

1.1.1.1 Les manifestations «physique»

Le traitement automatique de la langue a vu les prémices de son développement débiter dans le courant du XVII^e siècle. Ce développement est principalement dû à l'intérêt des mécanistes et des constructeurs qui cherchaient à reproduire le comportement humain, notamment à travers les automates (dont les premiers remontent à l'Ancienne Égypte⁵). La science mécaniste cherchait à expliquer à travers l'automate le fonctionnement du corps

⁴ En référence au livre de Jean-Pierre Sérés *Langages et machines à l'âge classique* (1995).

⁵ On peut contempler au Musée du Louvre un masque d'Anubis (Divinité accompagnant les défunts et protégeant les sépultures) possédant une mâchoire articulée actionnée par des fils cachés.

humain⁶ (Descartes, 1637). Et c'est dans le courant du XVIIe siècle que certains savants et philosophes vont essayer de comprendre et de reproduire artificiellement le langage en se basant sur les travaux du Père Mersenne⁷ (1588-1648). Le travail de ce dernier va provoquer un engouement de la part de nombreux savants et philosophes tel que Cordemoy (1626-1684) à qui l'on doit le *Discours physique de la parole* (1668), Dodart (1634-1707) qui publiera un certain nombre de traités sur la voix chantée entre 1700 et 1707 ou encore Ferrein⁸ (1693-1769).

Dans *Harmonie Universelle* (proposition XXI), Mersenne pense qu'il est possible de concevoir un orgue, qui adjoint à divers autres instruments de musique actionnés par des mécanismes hydrauliques (pour former les diverses harmonies), serait capable de reproduire tous les sons humains qui combinés produiraient toutes les syllabes et en conséquence toutes sortes de mots (Véronis, 2001). C'est le Danois Christian Gottlieb Kratzenstein (1723-1795)⁹ qui le premier fabriquera un orgue capable de prononcer cinq voyelles. Une représentation des résonateurs de Kratzenstein est montrée à la figure 1. Il est à noter que les formes représentées dans la figure 1 sont les formes des tuyaux de l'orgue de Kratzenstein.

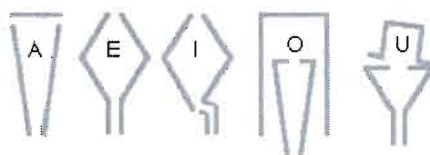


Figure 1 : Résonateurs de Kratzenstein.

⁶ L'un des buts premiers de cette recherche était de pouvoir s'affranchir du dogme de l'Église et de créer un savoir rationnel, objectif, transmissible et indépendant des opinions personnelles des chercheurs.

⁷ C'est le Père Marin Mersenne, à qui l'on doit également les premières lois de l'acoustique, qui envisageait la phonation «d'un point de vue articulatoire, acoustique et mécanique». *Harmonie Universelle* (1636-1637) (Proposition XLIII) dans Véronis, 2001.

⁸ Antoine Ferrein (1693-1769) est reconnu pour son travail sur la formation de la voix chez l'homme. Il est également l'un des trois co-signataires avec Georges-Louis Leclerc de Buffon (1707-1788) et Jean-Jacques Dortous de Mairan (1678-1771) du rapport remis à l'Académie royale des sciences le 9 juillet 1749, suite à la lecture par Jacob Rodrigue Péreire de son mémoire, relatant son art d'apprendre à parler aux sourds et muets de naissance.

⁹ Son orgue remporta le prix annuel de l'académie impérial de Saint Pétersbourg en 1780.

Durant la même période du XVIII^e siècle, le père Athanase Kircher (1601-1680) réalise une tête automate qui émet des sons, tandis que Hans Slottheim et Achille Langenbuscher, horlogers d'Augsbourg, fabriquent des instruments de musique jouant seuls. Ces diverses inventions sont à l'origine de la programmation actuelle. Chaque automate se mouvant sans l'intervention directe de son créateur, il a notamment fallu créer des systèmes de programmation et de commande. Ces programmes fonctionnaient principalement grâce à la science mécanique (Beaune, Doyon et Liaigre, 2008).

Toutefois, c'est suite à l'automate *Le joueur de flûte*¹⁰ de Vaucanson (1709-1782) que la production de cet art va connaître un très fort engouement. On notera la réalisation d'un automate écrivain¹¹ en 1760, par Frederik von Klaus (1724-1789), capable d'écrire 107 mots. Et c'est un peu avant 1780 qu'apparaissent les premiers automates phonéticiens, telles que la *tête d'airain* (1778), une machine capable de prononcer une phrase, ou les *Têtes parlantes*¹² de l'abbé Mical en 1783. Mais c'est la machine parlante (1779) du baron Wolfgang von Kempelen (1734-1804) qui fut l'automate phonéticien le plus remarquable (figure 2 et 3). Il est à noter que les essais du baron pour produire artificiellement des voyelles ont malheureusement perdu l'intérêt du public suite aux révélations sur son célèbre joueur d'échecs : *le turc*¹³.

¹⁰ *Le joueur de flûte* a été présenté en 1738 à l'hôtel de Longueville. Il s'agit d'un androïde assis de 1,50 m posé sur un socle de même dimension, et qui exécute rigoureusement les mêmes opérations qu'un joueur de flûte vivant. L'air actionnant l'instrument sort de la bouche de l'automate modulé par les lèvres. Un mécanisme fait bouger les doigts, en bouchant ou en dégageant les trous de l'instrument, ce qui produit les sons. Le joueur de flûte jouait douze airs différents, lentement ou rapidement, mais toujours avec justesse et précision. Parmi ces airs, on retrouve *Le Rossignol* de Blavet.

¹¹ Cet automate est toujours visible en Autriche, à Vienne.

¹² Présentée en 1783 à l'Académie des Sciences, elles sont composées de deux têtes. La première, par le biais d'un clavier cylindrique, permet l'oralisation d'un nombre déterminé de phrases ayant une prosodie correcte, alors que la seconde permet, par l'intermédiaire de son clavier, la reproduction d'un grand nombre de son et de ton de la langue française (Véronis 2001).

¹³ Le baron Wolfgang von Kempelen produisit dans en 1769 un automate capable de jouer aux échecs et qui s'avéra être une supercherie, le socle de l'automate contenant un complice.

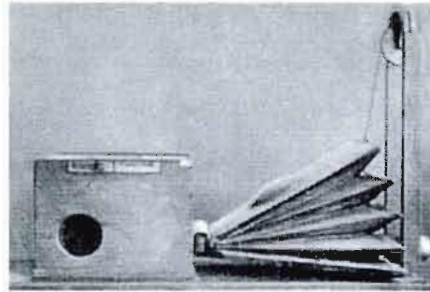


Figure 2 : Image d'une machine à parler du baron Von Kempelen

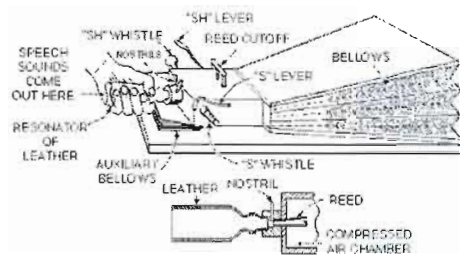


Figure 3 : Fonctionnement de la machine à parler du baron Von Kempelen d'après la description de Wheatstone (1802-1875) au XIXe siècle.

1.1.1.2 Les manifestations «abstraites»

Mais la passion des mécaniciens pour l'automatisation, concernant jusqu'alors des phénomènes physiques et mécaniques, va s'étendre dans des domaines qui semblent de prime abord bien moins évidents à automatiser, car appartenant à l'abstrait et à la logique : la grammaire et la sémantique.

Depuis longtemps, les grammairiens ont remarqué les régularités et la systématité du langage (Véronis 2001) et les automates vont modifier leurs approches de la langue et permettre la création de la mécanique des langues. Mais le passage des règles de construction aux mécanismes de construction des langues reste difficile à mettre en place (Séris, 1995).

Il se crée également au début du XVII^e siècle, un courant philosophique visant à créer des langues universelles entièrement artificielles (Séris, 1995). Cet engouement pour la mécanisation des langues va rejoindre un autre courant de recherche et d'études lancé durant le XVI^e siècle, celui de la cryptographie¹⁴. Les idées régissant cette dernière sont particulièrement proches de celles de la mécanisation des langues, les deux mettant en œuvre les mêmes types de mécanismes et de calculs. Ces calculs vont mener à l'émergence de la traduction mécanique et aux premières formes algorithmiques (Véronis, 2001).

Dans le même temps, des philosophes comme René Descartes (1596-1650) ou Gottfried Wilhelm von Leibniz (1646-1716) s'intéressent au projet de langues philosophiques. Ce projet, d'après Jean-Pierre Séris¹⁵ (Séris, 1995), se veut le «dénombrement de toutes les pensées des hommes et la découverte de l'ordre naturel de leur composition, à partir d'éléments primitifs». On ne parle plus alors de mécanisation de la grammaire, mais de mécanisation de la sémantique (bien que ce terme n'apparaisse qu'au début du XIX^e siècle (Véronis, 2001)).

En 1661, Dalgamo (1616?-1687) propose un modèle de langue basé sur des caractères universels. Il a pour idée que tous les hommes ont la même représentation des choses, mais une manière différente de les nommer. Il justifie son idée à l'aide des chiffres arabes, universellement utilisés et des langues à pictogrammes (comme le chinois). Il propose donc un système de signes qui représenteraient chacun une catégorie irréductible de la pensée humaine. À l'aide de règles automatisables, l'ensemble de ces signes formerait une langue artificielle et universelle (Dalgamo dans *Ars Signorum* (1661) cité dans (Cram et Maat, 1998)). Thomas Hobbes (1588-1679), compare la pensée à une méthode de calcul en précisant toutefois dans son ouvrage *De corpore* (1655) que le calcul peut ne pas porter que sur des nombres, mais également sur d'autres grandeurs¹⁶ (Duncan, 2009).

¹⁴ Dont les deux livres de «bases» sont la *Steganographia* (1518) de l'abbé de Würzburg et Jean Trithème, et le *Traicté des chiffres ou secrètes manières d'escrire* (1586) de Blaise de Vigenère (1523-1596) (Stern, 2008).

¹⁵ On trouve cette citation à la page 44 du livre de Jean-Pierre Séris. Citation également reprise dans Véronis (2001).

¹⁶ Dans son cours, Véronis cite un paragraphe du texte de *De corpore*, dans lequel Hobbes entend pouvoir effectuer des calculs sur des concepts, des actions, des mouvements, etc.

Les travaux mis en place quant au traitement du langage et son automatisation par les mécaniciens au cours du XVI^e et du XVII^e siècle ont apporté les fondements de ce qui trois siècles plus tard deviendra la base du traitement automatique du langage (Véronis, 2001).

1.1.2 Le XX^e siècle

Les progrès technologiques dans la première moitié du XX^e siècle sont relativement peu nombreux, tout du moins dans le domaine de l'informatique et du traitement de la langue. L'invention du téléphone¹⁷ (1876), qui devait à la base être un appareil auditif pour permettre aux personnes sourdes d'entendre le monde qui les entoure, va, durant les guerres suivantes permettre l'accélération du développement de l'électronique.

Bien que la cryptographie existait depuis plusieurs siècles, c'est le livre *Cryptographie militaire* (1883) d'Auguste Kerckhoffs (1835-1903), un linguiste, expliquant les diverses méthodes de cryptages et de codages qui va débiter le phénomène de décryptage des messages des camps adverses (Stern, 2008). Toutefois, durant la Première Guerre, peu de développements électroniques seront produits, malgré une volonté déjà présente de connaître par avance les mouvements de l'ennemi.

C'est durant la Seconde Guerre Mondiale que l'on a constaté l'émergence de l'électronique et son développement rapide. La nécessité pour les diverses alliances militaires de cette période d'être capable de déchiffrer les messages adverses tout en protégeant leurs propres messages, d'une éventuelle traduction, était vitale afin de connaître les mouvements des troupes ennemies et de les anticiper. C'est de cette volonté de décrypter les messages adverses afin de les comprendre qu'apparurent les premiers algorithmes de programmation. Le développement de ces premiers traitements algorithmiques de la langue est notamment

¹⁷ Par Alexandre Graham Bell (1847-1922).

dû à Alan Turing¹⁸ au centre de décryptage de Bletchley Park (Hodges, 2000). Mais c'est en 1945 que le mathématicien américain Warren Weaver imagine l'utilisation des calculateurs pour le décryptage et la traduction (Véronis, 2001).

C'est à partir des recherches et des avancées réalisées sur le décryptage que va naître, en 1946, le mouvement du Traitement Automatique de la Langue sous l'appellation de Traduction Automatique (T.A.). Il s'agit alors de décoder un texte puis de le traduire dans la langue de l'utilisateur. La T.A. appartient alors aux informaticiens.

Cet effort de guerre, débuté par la recherche en cryptographie, va se poursuivre à la fin de la Seconde Guerre Mondiale en raison des courses à l'armement et à la conquête spatiale générées par la Guerre Froide. Cette époque fut elle aussi à l'origine de nombreux développements technologiques.

Du côté du traitement de la langue et de la traduction, les scientifiques de l'époque voient et traitent la langue comme s'il s'agissait d'un code sans connaissances linguistiques. Les résultats des traductions sont généralement médiocres (Abeillé, 2008).

Mais la programmation algorithmique progresse et souvent plus rapidement que celle des ordinateurs. Elle se voit ralentie par le besoin de matériel de plus en plus puissant pour les calculs ce qui va accélérer le développement électronique.

L'apparition de l'écran, en 1951, va grandement faciliter l'implémentation des algorithmes. Cette simplification de la programmation algorithmique va attirer certains linguistes qui commencent à entrevoir les diverses possibilités offertes par les ordinateurs dans l'analyse de la langue. C'est le cas de Yehoshua Bar-Hillel (1915 - 1975), linguiste, philosophe et informaticien qui, en 1952, intègre le M.I.T. (Massachusetts Institute of Technology) (Véronis, 2001).

Par ailleurs, en 1955, Noam Chomsky entre également au MIT et débute un travail sur les automates et le langage. Il y est alors collègue avec Bar-Hillel et Marvin Minsky

¹⁸ Alan Turing (1912-1954) fut à l'origine de la programmation algorithmique, créateur de la Machine de Turing, en 1936, qui permet d'enquêter sur l'étendue et les limites de ce que l'on peut programmer. On utilise encore son algorithme à l'heure actuelle en informatique théorique. Il proposera également le Test de Turing qui a pour but, encore aujourd'hui, de déterminer le degré d'intelligence d'un programme d'Intelligence Artificielle (1950). Pour plus de détails, je vous réfère à la réédition américaine du livre d'Andrew Hodges (2000). Hodges, *Alan Turing : the Enigma*.

(informaticien et chercheur en sciences cognitives). C'est ce dernier qui, avec John McCarthy et Claude Shannon¹⁹, organisera, en 1956, la conférence de Dartmouth où, ni Bar-Hillel, ni Chomsky, ni aucun spécialiste de la traduction ne seront conviés bien que l'une des premières volontés des informaticiens était de permettre la communication entre l'Homme et la Machine (Véronis, 2001). C'est durant cette conférence que la programmation et les recherches algorithmiques seront canalisées et orientées vers un même but : la création de l'Intelligence Artificielle

En Europe, l'Association pour l'étude et le développement de la Traduction Automatique et de la Linguistique Appliquée (A.T.A.L.A.) voit le jour en 1959 et commence, en 1960, la publication de la revue Traduction Automatique qui publie l'ensemble des parutions du domaine. C'est à la même époque que le Traitement Automatique de la Langue (T.A.L.) devient une discipline autonome et se détache de la T.A. pour devenir une branche de la linguistique : la linguistique computationnelle.

En 1960, aux États-Unis, Bar-Hillel va écrire un rapport (Bar-Hillel, 1960) relatant des « *difficultés énormes que pose la traduction, tant sur le plan technologique que linguistique* » (Véronis, 2001). Il sera le premier à parler du besoin, pour la machine, de posséder un domaine de connaissance décrivant le monde. Toutefois, il est le premier à reconnaître l'impossibilité d'intégrer de tels domaines dans les machines de l'époque.

En 1966, le rapport A.L.P.A.C. (Automatic Language Processing Advisory Committee), commandé par l'administration américaine à la suite du rapport de Bar-Hillel, va entraîner l'arrêt des financements et la fermeture de nombreux laboratoires de recherches en T.A. aux États-Unis malgré la réalisation de progrès importants (Véronis, 2001). Néanmoins, quelques laboratoires au sein de certaines universités vont continuer la recherche²⁰.

À l'époque, les principaux courants de la linguistique appliquée sont assez éloignés de l'I.A.. Ils se concentrent davantage sur l'analyse de données (analyses sémantiques, analyses des structures syntaxiques, etc.) que sur le dialogue Homme-Machine imaginé par les

¹⁹ Claude Shannon qui travailla de 1941 à 1972 au Bell Labs avant d'intégrer, en parallèle, le MIT de 1958 à 1976. Il est notamment connu pour sa théorie de l'information.

²⁰ On notera notamment le cas du laboratoire de la Brigham Young University dans l'UTAH qui continuera à travailler sur la traduction de la Bible, sous le financement de l'Église Mormone.

groupes de recherche sur l'I.A.. Mais, malgré les promesses de l'informatique quant aux traitements des données, les possibilités de stockage, trop réduites à l'époque, ou le manque de mémoire pour les processus ne permettent pas de réaliser l'analyse des textes longs et des corpus. Toutefois, l'analyse de petits textes permet d'opérationnaliser un grand nombre d'hypothèses quant aux règles qui régissent les langues. Et c'est dans la vérification de ces règles que la génération de texte commence à se développer. Afin de vérifier si les règles postulées dans leurs hypothèses sont correctes, les chercheurs en linguistique créent avec l'aide d'informaticiens les premiers générateurs (Friedman, 1969) au début des années soixante-dix. C'est la naissance de la Génération Automatique de Texte (G.A.T.), dont le but est de générer aléatoirement des phrases afin de pouvoir les analyser pour vérifier leur grammaticalité et valider les hypothèses posées (Danlos, 1985). Pourtant, ce n'est que vers la fin des années soixante-dix que la G.A.T. commencera son réel développement (Danlos, 1990).

Alors que la recherche en T.A. est fortement ralentie aux États-Unis, la situation en Europe est différente. La construction de la Communauté Européenne, en marche depuis la fin de la Seconde Guerre Mondiale, nécessite des besoins constants en traduction. Et bien qu'elle subisse elle aussi les conséquences du rapport de l'A.L.P.A.C., la recherche européenne ne s'arrête pas. C'est d'ailleurs par la demande de l'Union Européenne, en 1975²¹, que les travaux et la recherche sur la traduction automatique connaissent un nouvel engouement, principalement avec la création et le développement du système SYSTRAN²².

Malgré quelques résultats impressionnants, tels que le programme ELIZA de Weizenbaum²³ en 1966 qui simule un dialogue avec un psychanalyste, ou le programme SHRLDU de Terry Winograd qui comprend les ordres donnés lors d'un dialogue, les publications de Weizenbaum (1976) et de Dreyfus et Dreyfus (1986) vont avoir des répercussions presque similaires au rapport A.L.P.A.C. (Véronis, 2001) sur la recherche sur l'I.A.. Mais contrairement à la T.A., la recherche sur l'I.A. ne sera pas arrêtée, mais redirigée et intégrée dans la recherche sur les systèmes informatiques (Véronis, 2001). Les

²¹ Notamment due à l'entrée du Danemark, de l'Irlande et du Royaume Uni dans la Communauté Européenne en 1973.

²² Il s'agit d'un programme de compréhension du langage naturel développé au MIT par Winograd entre 1968 et 1970.

²³ Mathématicien et Informaticien allemand. Créateur du programme ELIZA (<http://jerz.setonhill.edu/if/canon/eliza.htm#Interaction>).

informaticiens estiment que le développement de systèmes moins complexes, tels que les systèmes experts²⁴, pourrait également permettre la création d'une intelligence artificielle.

1.1.2.1 Le domaine de connaissance

L'un des avantages du système expert est d'être relativement facile à implémenter, dans la mesure où un système expert est un ensemble de petits programmes « simples » travaillant en collaboration les uns avec les autres. Toutefois, comprendre le fonctionnement d'un système expert afin de le rendre encore plus efficient est une des priorités des informaticiens. Mais leurs propres générateurs n'ont pas une assez grande connaissance de la langue pour produire des textes exploitables.

Les informaticiens redécouvrent donc les difficultés décrites d'une part par Bar Hillel (Bar-Hillel, 1960) et d'autre part par les linguistes. Les principaux problèmes reposent sur les propriétés des langues et les situations de communication.

C'est donc au début des années quatre-vingt que les informaticiens demandent aux linguistes d'intégrer leurs équipes afin de développer des outils plus efficaces dans leur recherche de communication avec la machine. Mais peu de linguistes accepteront la main tendue, la plupart préférant se concentrer sur l'analyse automatique de corpus de plus en plus importants.

Les premiers développements de générateurs de textes adaptés sur les systèmes experts soulèvent un nouveau problème d'envergure. Afin de tester leurs générateurs, les linguistes-informaticiens les intègrent dans des systèmes experts. Le but était de comprendre le fonctionnement interne du programme. Le système est chargé de générer un texte explicatif de l'ensemble de ses actions afin que les chercheurs puissent les suivre. L'hypothèse posée était que la compréhension du déroulement pourrait faciliter l'implémentation des futurs programmes et de l'I.A.. Cependant, même si un grand nombre de tâches est clairement explicité par le générateur dans le texte généré et que l'ensemble est

²⁴ Un système expert est un logiciel capable de répondre à des questions à partir de faits et règles connus en effectuant des raisonnements sur ses connaissances.

grammaticalement et lexicalement correct, certains points de la génération restent incompréhensibles. En effet, les textes générés semblent contenir de sérieux problèmes de sémantique et des actions inconnues que les programmeurs ne peuvent expliquer, Swartout (1981) cité dans Danlos (1985).

Dans les traces de Bar-Hillel, Swartout (1983) estime que le problème ne vient pas du programme de génération, mais du système expert²⁵. Son explication provient du fait que l'exécution d'un programme nécessite l'exécution d'autres sous-programmes contrôlés par le système lui-même et non pas par le logiciel. Pour prendre un exemple simple, lorsque l'on demande à un logiciel d'afficher une fenêtre, ce dernier appelle le système, et c'est ce dernier qui va se charger de l'ouverture et de l'affichage, ne renvoyant au logiciel que la fenêtre ouverte. Tout le code pour l'ouverture de la fenêtre est réalisé par un programme du système auquel le logiciel n'a pas accès. Le logiciel n'a donc pas la possibilité d'expliquer comment la fenêtre a été ouverte. C'est ce qui arrive dans les textes générés par les systèmes experts. Ces derniers n'ayant pas accès aux programmes extérieurs contrôlés par le système d'exploitation, ils ne possèdent pas tous les éléments pour expliquer leur fonctionnement. Swartout (1983) appelle cet ensemble de programmes parallèles indépendants, mais vitaux, le domaine de connaissance du logiciel. Il propose de l'intégrer dans le module de génération afin de pouvoir expliquer les parties manquantes des textes produits.

1.1.2.2 La représentation du domaine de connaissance

La découverte du lien entre programme et domaine de connaissance va entraîner de grands changements dans le traitement automatique des données, aussi bien en génération qu'en analyse. Maintenant apparaît le problème de la représentation de ces connaissances. Comment intégrer cette représentation ? Quels sont les formalismes à utiliser pour les représenter ? Autant de questions qui surgissent de ce besoin pour le système de connaître le monde dans lequel il évolue.

²⁵ Pour plus de clarté dans mon exemple, je parlerai du système expert sous l'appellation *logiciel* et du système d'exploitation de l'ordinateur (par exemple Windows, OSX, Ubuntu,...) sous l'appellation *système*.

Ces systèmes de représentation du domaine de connaissance vont être modélisés sous trois différentes formes : une modélisation linguistique, mathématique ou cognitive (Abeillé, 2008).

1.1.2.2.1 Modélisation linguistique

Pour la modélisation linguistique, le but est de représenter tous les niveaux (et sous-niveaux de la langue). Il s'agit donc de segmenter la représentation linguistique en fonction des niveaux de la linguistique. On aura ainsi une segmentation phonétique, phonologique, morphologique, syntaxique, sémantique, pragmatique. Plusieurs théories de segmentation ont été utilisées, telles que les grammaires génératives ou plus récemment les grammaires d'unification (Abeillé, 2007). Ces dernières connaissent une plus grande utilisation du fait de la prise en compte du lexique et de la sémantique.

1.1.2.2.2 Modélisation mathématique

Le modèle mathématique, quant à lui, ne tient pas compte de la sémantique. À l'aide de corpus préanalysé et d'études stochastiques²⁶, on obtient des résultats de traitements intéressants, mais peu fiables au niveau linguistique (Abeillé, 2008). Le principal problème de cette approche est la nécessité d'avoir accès en permanence à des corpus énormes dont il faut recalculer les probabilités dès que le type de texte à produire ou que le domaine d'application change.

On retrouve également dans les modèles mathématiques les automates à états finis. Ces automates permettent de traiter rapidement et de façon déterministe l'analyse lexicale ou syntaxique de l'information dans des corpus de grande taille. En revanche, l'étude effectuée n'est que locale, cela signifie que l'on ne dispose que de peu d'informations sur le

²⁶ Approche basée sur les probabilités. Dans ce cas, on va calculer la probabilité d'un mot en fonction des probabilités des mots précédents.

contexte de la partie analysée. Néanmoins, cette méthode permet de rapprocher le modèle mathématique et le modèle linguistique, les automates à état finis se voulant équivalents à la grammaire de type 3 de Chomsky (Kahane, 2008).

1.1.2.2.3 Modélisation cognitive

Les modèles cognitifs vont tenter de reproduire les comportements langagiers. Ces modèles ont un but qui relève du test de vérification d'hypothèses sur le fonctionnement de la parole (et de l'humain en général).

L'ensemble de ces modèles a l'avantage de permettre au programme de faire des inférences²⁷ sur le domaine de connaissance. De nombreux essais ont donc été réalisés, et ce dans divers langages²⁸. Chacun de ces langages a été pensé et créé au fur et à mesure de la complexification des requêtes utilisateur et surtout dans le but de la réalisation de ces requêtes. Parmi ces langages, on retrouve :

- la logique propositionnelle (Lepage, 2001) : il est rapidement apparu que les inférences réalisables étaient limitées, ce langage n'admettant ni les relations, ni les variables.
- la logique de prédicats (Lepage, 2001) : ce langage basé sur la logique propositionnelle a pour avantage de permettre les relations entre les unités ainsi que les variables.
- le réseau sémantique : un moyen pour relier plusieurs concepts à l'aide de la relation sémantique (Cogsweb Project). Ces relations sont généralement hiérarchisées. Toutefois, le réseau sémantique ne permet pas la formalisation de son contenu.

²⁷ Inférence : substantif féminin, opération logique par laquelle on admet une proposition en vertu de sa liaison avec d'autres propositions déjà tenues pour vrai, *Le petit Robert*.

²⁸ L'utilisation du terme langage est abusive dans ce cas. Il s'agit d'une mauvaise traduction du mot anglais « language ». En effet, on parle couramment de langage informatique alors qu'il conviendrait mieux parler de formalisme informatique.

- le graphe conceptuel (Sowa, 1983) : possède un aspect formel qui le rapproche de la logique des prédicats, dans son fonctionnement.
- les frames : représentation des connaissances sous la forme de schéma (Minsky, 1975).

Il en ressort aujourd'hui que c'est en combinant plusieurs modèles que l'on obtient les meilleurs résultats, et notamment en unifiant modèles linguistiques et modèles mathématiques (Abeillé, 2008).

C'est d'ailleurs l'unification des frames et des réseaux sémantiques qui va donner naissance aux ontologies. Une ontologie est une représentation des connaissances sous forme de concepts reliés entre eux par des liens sémantiques.

La première ontologie à voir le jour est 'wordnet' mise au point par Christiane Fellbaum et Georges Miller en 1985 (Miller et Fellbaum, 2007). Lors de sa mise en service, 'wordnet' était une ontologie programmée sous forme de thesaurus qui regroupait des informations sur le lexique de la langue anglaise. Aujourd'hui, elle comporte, en plus de l'anglais, au moins une dizaine de langues différentes (Espagnol, Allemand, Français, Tchèque, etc.) (Rastier, 2004). Je reviendrai plus longuement sur les ontologies dans le point 1.3.2 Ontologie et Protégé.

Les années quatre-vingt-dix marquent un tournant dans le développement du traitement automatique du langage et cela grâce à la sortie grand public de logiciels tels que les logiciels de traitement de texte, les logiciels de reconnaissance et de synthèse vocale, l'arrivée des correcteurs orthographiques pour les traitements de textes et tous les outils web (moteur de recherche, traducteur automatique).

Ces dernières années, le développement du Web sémantique²⁹ a permis de nouvelles avancées dans le T.A.L. ainsi que l'émergence de nouveaux défis. Pourtant, ces avancées ne concernent pour la plus grande part que l'analyse des données et très peu la génération.

²⁹ Le Web sémantique fournit aux données une structure commune qui leur permet d'être partagées et réutilisées à travers diverses applications.

1.2 Génération

Générer automatiquement du texte signifie faire écrire par un logiciel un texte. Mais ce texte doit obéir à certaines contraintes, notamment être une production acceptable linguistiquement. Le but ultime des programmeurs de générateur est bien évidemment de faire passer à leurs algorithmes le test de Turing. Cela n'entre pas dans mes objectifs directs, ma motivation première étant de réussir à générer des phrases acceptables. Il reste tout de même dans la génération automatique, le problème de la volonté communicationnelle. En effet, l'acte d'écriture est un acte de communication mais le texte généré automatiquement n'entre pas, de prime abord, dans un vrai cadre discursif (Bronckart, 1994).

Depuis la création du T.A.L., les recherches se sont principalement tournées vers l'analyse d'une production humaine et bien que les premiers générateurs datent de la fin des années cinquante, la génération de textes n'a réellement commencé à se développer que vers la fin des années soixante-dix (Danlos, 1990). Ce développement tardif serait dû au fait que la génération a des besoins beaucoup plus complexes que ceux nécessités par l'analyse, celle-ci faisant, généralement, partie du processus de génération.

En effet, la génération de texte, contrairement à l'analyse, est un acte qui se produit en deux temps distincts que nous développerons plus en détail dans la suite de ce mémoire : la génération profonde et la génération de surface.

L'intérêt de la génération de texte peut être abordé de deux points de vue différents. Le premier point de vue est celui des linguistes qui ne désiraient générer du texte que pour valider (ou invalider) leurs théories sur les règles de constructions de la langue. Le second point de vue est celui des informaticiens, point de vue sur lequel nous reviendrons dans la suite de ce texte.

Le premier point de vue est celui qui fut le plus utilisé afin d'effectuer la génération de textes depuis ses débuts jusqu'au milieu des années quatre-vingt. Il s'agit d'une solution de génération «facile» où cette dernière est effectuée à l'aide de textes préenregistrés.

Pour chaque cas que l'on souhaite générer, la machine va rechercher dans ses bases de données le texte préenregistré correspondant. Cette méthode s'avère efficace tant que le nombre de cas n'est pas trop grand. Dans le cas contraire, des problèmes de gestion de la mémoire et du temps de recherche du bon texte font leur apparition. De plus, chaque évolution du logiciel entraîne une mise à jour des textes à produire, ce qui peut vite devenir problématique. Afin de mettre fin à ce problème de maintenance, les linguistes vont modifier leur générateur en utilisant des textes préenregistrés à variables. Il s'agit du même procédé que celui utilisé par les informaticiens avec la génération par phrases à trous. Certains mots du texte préenregistré sont remplacés par des variables. Chaque variable possède plusieurs solutions parmi lesquelles le programme choisira, en fonction de la demande de l'utilisateur, la solution la plus adaptée à ce qu'il doit produire (Danlos, 1990).

Cette méthode de génération va permettre aux linguistes de développer des règles de production et des grammaires afin d'en vérifier la validité. Il s'agit d'une méthode de génération de phrases aléatoires utilisée pour vérifier des grammaires non contextuelles (Yngwe, dans (Sabah, 1989)) dans un premier temps, puis les grammaires transformationnelles (Friedman, 1969). L'ensemble des phrases aléatoires produites forment des textes qui même s'ils sont cohérents, syntaxiquement et sémantiquement, ne sont pas des textes fluides.

Le but de la génération n'est pas seulement de produire du texte, mais de produire du texte dont les caractéristiques langagières et structurelles sont comparables à la production écrite d'une personne. En d'autres termes, on veut que le programme se comporte dans sa production comme un humain³⁰. Ce qui n'est pas le cas avec la génération à variables (Bateman, 2002).

Le second point vu, celui des informaticiens, a un but quelque peu différent de celui des linguistes. Effectivement, les programmeurs, désireux de produire H.AL., sont à la recherche de nouveaux formalismes théoriques afin que la machine puisse produire du texte

³⁰ Le rêve des chercheurs en I.A. a toujours été la création d'un ordinateur tel que H.AL., élaboré, en 1968, par Arthur C. Clarke et Stanley Kubrick dans le film de ce dernier : *'2001, l'odyssée de l'espace'*. H.AL. est un acronyme pour *'Heuristically Programmed Algorithmic computer'*.

'humain'. Il s'agit alors, pour eux, de modéliser les processus cognitifs humains sous-jacents à la production de texte.

Les tentatives de modélisation de ces processus vont amener les chercheurs à réfléchir sur les méthodes de production de texte et vont rapidement faire émerger des questions fondamentales pour la génération (Sabah, 1989).

1.2.1 Les questions fondamentales

Les questions fondamentales qui ressortent de ces réflexions sont aux nombres de trois :

- le Quoi dire ?
- le à Qui le dire ?
- le Comment le dire ?

Il peut être pertinent de se poser également la question du quand le dire. Toutefois, cette dernière question trouve sa place principalement dans la génération de dialogue, mais elle n'est que peu pertinente dans la génération de texte.

Pour générer un texte, il faut donc être capable de répondre à ces trois questions. La première question, le quoi dire, représente la première étape de la génération de texte : la génération profonde, alors que les deux dernières représentent la seconde étape : la génération de surface.

La génération profonde va correspondre à la prise de décision de la part du système. C'est pendant cette partie de la génération que le système va choisir les informations à fournir à l'utilisateur dans le texte à générer. Toutefois, les informations choisies seront présentes dans le texte sous une forme différente de celle où elles apparaissent dans la génération profonde. En effet, il s'agit à cet instant d'effectuer une représentation langagière du contexte (Bronckart, 1994), en général, à l'aide de deux outils. Le premier outil étant celui

qui va choisir les connaissances à transmettre, et le second, celui qui va les structurer dans le 'futur' document (Ponton, 1997). Cette prise de décision sur le contenu du texte à générer concorde avec la question du quoi dire. C'est effectivement cette période de décision qui remplit cette tâche et répond à cette question.

La génération de surface, quant à elle, va associer des unités lexicales et une syntaxe adaptée aux idées/concepts retenu(e)s lors de la génération profonde. Ce module va effectuer la conceptualisation du message, et notamment choisir quel explicite le texte devra contenir et par là même, quel implicite. C'est à ce niveau qu'apparaît le module de connaissance du domaine et du monde. C'est ce module qui va permettre de mettre le texte en forme en choisissant le lexique et la syntaxe adaptés à la production voulue. Il s'agit de la textualisation du message (Bronckart, 1994).

Toutefois, le message généré ne se fait pas dans le même cadre que l'écriture classique d'un texte par un humain. En effet, une communication écrite humaine possède un but que ne possède pas une communication écrite générée par un logiciel (sortie du cadre de l'enseignement à distance tel que l'e-learning ou les STIs). Dans une communication écrite humaine, le texte a un but et est écrit en fonction de la représentation que l'auteur se fait de son lecteur. Alors que dans le cadre d'une génération automatique, le logiciel a rarement une représentation de son lecteur. Bien qu'ayant un but, la communication devient, dans ce cas, unidirectionnelle.

Trois questions sont donc clairement définies dans la littérature, cependant, il en est une qui n'est pas, ou peu, soulevée. Il s'agit de la question du pourquoi écrire (Culioli, 1999). Cette question a son importance dans tout acte langagier et devient donc une question aussi pertinente dans la génération automatique que les trois autres questions soulevées précédemment. Néanmoins, n'ayant pas trouvé d'informations concernant ce point, je ne peux y répondre actuellement, mais je proposerai tout de même mon idée de réponse dans la suite de ce rapport (cf: 1.2.3.1 Le 'à qui dire?').

1.2.2 La génération profonde

Générer du texte est un domaine des plus intéressants dans la linguistique appliquée, mais encore faut-il savoir quoi générer (Danlos, 1990). Générer du texte pour générer du texte n'a pas grand intérêt. La génération doit correspondre et surtout répondre à une demande de la part de l'utilisateur. Et c'est en génération profonde que cette question du 'quoi générer' trouve sa réponse.

Le générateur de texte fonctionne généralement en association avec un analyseur sémantique, afin de comprendre les requêtes qui lui sont faites. Pour cela il existe plusieurs méthodes d'analyse possibles. Parmi celles-ci se retrouvent les réseaux de transitions (Woods, 1970) ou les réseaux discriminants (Ponton, 1997).

Le réseau de transition va étudier la demande de l'utilisateur sous forme d'un automate fini. Toutefois, l'analyse fournie reste limitée à des structures syntaxiques données (par le programmeur) et ne contenant pas d'informations contextuelles. Cette limitation est due au fait que les automates finis ne possèdent pas de mémoire directe³¹. Des méthodes pour les rendre plus efficaces ont été développées, telles que les réseaux de transitions augmentées (ou ATN) et les réseaux de transitions récursifs (Faure et al., 2005).

Pour sa part, le réseau discriminant possède une mémoire dans laquelle il stocke l'ensemble des cas d'analyses rencontrés. Cela lui permet lorsqu'il bloque sur une structure particulière de vérifier s'il n'a pas déjà résolu un cas identique précédemment (Faure et al., 2005).

Une fois l'analyse effectuée et la demande de l'utilisateur établie, l'analyseur va transmettre au générateur cette dernière, soit ce sur quoi il doit générer.

Le programme va alors calculer en fonction de ses algorithmes la meilleure façon d'apporter à l'utilisateur l'ensemble des informations pertinentes pour ce dernier. Il s'agit ici de relever l'ensemble des informations à fournir à l'utilisateur sur le thème demandé. Ces informations ne seront pas alors rédigées sous la forme d'un texte, mais sous la forme d'une

³¹ Les automates finis n'ont aucune possibilité de stocker des informations c'est pourquoi, le programmeur lui fournit un ensemble de structure syntaxique qui serviront de point de comparaison.

structure informationnelle. Cette structure, en plus de contenir les informations nécessaires, inclura également la hiérarchisation de ces dernières. La génération de surface n'aura donc plus qu'à reprendre cette structure afin de la transformer en texte cohérent et utilisable par l'utilisateur (Sabah, 1989).

La génération profonde correspond donc à la question du 'quoi dire'. Plusieurs méthodes de production de la structure profonde existent. Parmi ces méthodes, on retrouve principalement la méthode sens-texte, la méthode par arbres adjoints ou les grammaires de traits.

Les méthodes de génération par arbres adjoints et par les grammaires de traits produisent à la fois de la génération profonde, mais également de la génération de surface, principalement lors de génération de phrase. Cela est dû au fait que ces deux méthodes ne contiennent pas que des informations syntaxiques sur la structure de la phrase à produire.

1.2.2.1 La génération sens-texte

La théorie sens-texte a été développée par Igor Mel'čuk en collaboration avec d'autres chercheurs russes dans la fin des années soixante. Néanmoins, elle ne deviendra populaire que tardivement (au début des années deux mille).

Les modèles sens-texte sont des modèles formels. En cela, ils ont pour but de représenter des énoncés linguistiques, mais également de permettre l'implémentation de règles de manipulation de ces représentations.

De plus, la théorie sens-texte doit pouvoir décrire l'ensemble des langues. Pour Alain Polguère (1998), cité dans (Tremblay, 2009), la théorie sens-texte est une théorie reposant sur des principes généraux s'appliquant à toutes les langues et non à une seule langue spécifique. Elle permet toutefois de créer des modèles spécifiques à chaque langue.

Alain Polguère (1998), cité dans (Tremblay, 2009), pense les modèles sens-texte comme des modèles computationnels. Ils sont donc implémentables et peuvent servir dans des applications informatiques nécessitant des connaissances lexicales et grammaticales.

Des recherches sont également menées afin de coupler la théorie sens-texte et la génération par grammaire d'arbres adjoints (Kahane et Lareau, 2005).

La théorie sens-texte est une théorie faisant partie de celles développées dans le cadre de l'analyse et de la génération automatique. Cependant, je ne m'attarderai pas sur sa description, celle-ci ne correspondant (Tremblay, 2009) aux objectifs de mon projet.

1.2.2.2 La génération par grammaire d'arbres adjoints

On connaît ces grammaires sous le nom de T.A.G. pour Tree Adjoining Grammar. Il s'agit d'un modèle mathématique mis en place par Aravind K. Joshi en 1975.

Contrairement à beaucoup de modèles génératifs, ce modèle n'est pas basé sur des règles de réécriture, mais sur des arbres élémentaires (Figure 4). Chaque arbre élémentaire est une unité de base associée à un item lexical. Chaque unité est combinée avec les autres par deux opérations possibles : la substitution et l'adjonction (Abeillé, 2007).



Figure 4 : Arbres élémentaires pour la phrase : 'Jean dort'

L'avantage de cette grammaire est de coupler le choix syntaxique et le choix lexical. En effet, le choix d'un mot va entraîner automatiquement le choix d'arbres élémentaires possédant leurs propres contraintes syntaxiques. Il s'agit d'ailleurs du principal intérêt de ce type de grammaire. Ainsi l'on est en possession d'une description lexicale automatiquement

associée à une description syntaxique, de ce fait, les grammaires d'arbres adjoints ne permettent pas d'avoir des «règles syntagmatiques indépendantes des propriétés combinatoires des éléments lexicaux» (p.261, (Abeillé, 2007)).

Un autre avantage de cette grammaire est que les relations de dépendances sont immédiatement visibles et peuvent, pour certaines se noter directement entre les noeuds concernés. Pour Anne Abeillé (2007), cette représentation a un intérêt certain au niveau du codage, car cela permet une simplification des calculs d'unification, mais d'un autre côté, la simplification du système de traits syntaxiques associé oblige à la création d'un grand nombre d'arbres élémentaires différents pour les diverses représentations d'un même item lexical.

Ce type de génération connaît un certain engouement (Kahane et Lareau, 2005) même si les productions sont assez peu nombreuses. Ce fait peut s'expliquer par la pauvreté sémantique des arbres de dérivation et des arbres élémentaires. Toutefois, dans son ouvrage sur les grammaires d'unification, Anne Abeillé (2007) écrit que «des travaux récents ont conduit à développer des arbres élémentaires avec des structures de traits sémantiques explicites associées».

1.2.2.3 La génération par grammaire de traits

Les grammaires de traits, aussi connues sous le nom de grammaires syntagmatiques guidées par les têtes (ou Head-driven Phrase Structure Grammar - H.P.S.G.) ont été développées au début des années quatre-vingt par Carl Pollard et Ivan A. Sag (Pollard et Sag, 1994). Le premier ouvrage de Pollard et Sag sur la HPSG date de 1987, mais dans ce second ouvrage, les auteurs développent les nouvelles directions prises notamment au niveau de l'organisation des structures de traits.

La HPSG est un formalisme grammatical qui revendique l'héritage de divers cadres théoriques, celui des grammaires syntagmatiques généralisées (GPSG), des grammaires catégorielles (GC) et des grammaires lexicales-fonctionnelles (LFG). Il ne s'agit pas d'un modèle dérivationnel, car toutes les informations sont traitées sur un même niveau de

représentation, contrairement à la théorie du gouvernement-liage de Chomsky, qui traite les relations entre les éléments à l'aide du mouvement ou des transformations (Blache, 2007). On analyse donc la syntaxe, le lexique et la sémantique dans un même temps et sur un même plan de description, et cela grâce à l'intégration dans la structure de différents types d'informations.

Les grammaires de type HPSG sont des grammaires enrichies d'un mécanisme d'unification. Toutes les règles sont représentées par une matrice d'attribut-valeur. Dans l'exemple d'Abeillé (2007), Figure 5, le trait 'SYNTAXE' correspond un attribut et '[CATÉGORIE P]' correspond à la valeur de cet attribut.

Comme on peut le constater sur la Figure 5, chaque structure de trait peut être récursive, ce qui apparaît avec l'attribut 'BRANCHES' qui a pour valeur un ensemble de structures de trait. Ainsi la première valeur de 'BRANCHES' est un attribut 'TÊTE' qui a pour valeur une autre matrice attribut-valeur.

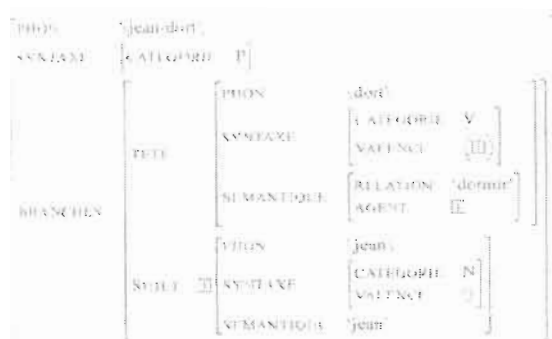


Figure 5 : Structure simplifiée de la phrase 'Jean dort'

L'un des points forts de ce type de grammaire est, comme je le signalais plus haut, l'intégration dans sa structure de différents types de connaissances, qu'il s'agisse de connaissances syntaxiques, lexicales, phonologiques, sémantiques ou pragmatiques, même si la plupart des connaissances représentées dans la structure de trait concernent principalement les aspects syntaxiques, sémantiques et lexicaux.

Les grammaires HPSG contiennent également un typage des informations. Le typage permet d'avoir pour chaque type des traits appropriés. Il s'agit d'une relation fondamentale entre trait et type : «*Un trait est approprié pour un type s'il fait partie des valeurs possibles pour ce type.*» (Blache, 2007). Pour reprendre l'exemple de Blache (2007), cela signifie que le trait *VFORM* est approprié pour le type *verbe*, mais pas pour le type *nom*. Le type permet donc d'associer un ensemble de traits spécifiques à chaque partie de la structure de la matrice. Les types en HPSG transmettent également les notions d'héritage de traits. Un trait approprié à un type donné *t* sera également approprié au sous-type de *t*.

L'intérêt d'utiliser les types permet de vérifier la cohérence de la grammaire. En effet, il est impossible d'associer un trait à un type qui ne lui correspond pas. Ainsi l'utilisation d'une même structure de traits typés pour représenter le lexique, la syntaxe, la sémantique et même les phrases de la langue, permet de «dépasser les limites des arbres syntagmatiques de plusieurs façons : par un typage différencié des différentes branches, et par des règles d'ordre des mots qui peuvent ordonner des constituants de niveaux différents» (p192, (Abeillé, 2007)).

Dans la Figure 6 (il s'agit ici d'une structure de Pollard et Sag (1994) traduite dans Blache (2007)), on peut constater la complexité de la structure d'un simple syntagme. Le trait '*SYNSEM*' est un trait qui, depuis la seconde version de la HPSG, regroupe l'ensemble des traits liés aux informations syntaxiques et sémantiques qui décrivent le constituant. La majorité des traits appartiennent à ce trait. Seuls les traits '*FILS*' (qui marque la hiérarchie) et le trait '*QMEM*' (qui marque la quantification) n'appartiennent pas '*SYNSEM*'. On peut néanmoins remarquer que le trait '*FILS*' contient le trait '*SYNSEM*'. L'explication est simple, le trait '*FILS*' est un élément lexical qui a ses propres valeurs syntaxiques et sémantiques qui sont les valeurs de ce trait. Le trait '*FILS-TÊTE*' représente comme son nom l'indique la tête du constituant, ici du syntagme nominal, alors que le trait '*FILS-SPR*' représente, quant lui, le spécificateur sous-catégorisé par le constituant.

les éléments représentés. C'est d'ailleurs la systématisation des informations dans une structure unique qui rend cette grammaire si intéressante.

De plus, le modèle HPSG a réussi à modéliser un grand nombre de phénomènes linguistiques, et notamment des phénomènes syntaxiques dans diverses langues. Bien que pour certaines langues, telles que l'allemand, des adaptations ont dû être effectuées, le modèle a l'avantage d'être stable pour la plupart des langues (Abeillé, 2007).

Ce modèle a déjà été utilisé pour la génération, notamment avec l'algorithme mis en place par Shieber et al. (1990). Il continue d'ailleurs d'être utilisé et développé, notamment au niveau de l'intégration d'éléments phonologiques, morphologiques et discursifs (Abeillé, 2007).

De nombreux travaux traitent des HPSG et certains concernent la création d'une ontologie linguistique basée sur une grammaire HPSG. La base de cette ontologie est une ontologie linguistique se nommant GOLD. Il s'agit d'une ontologie qui se veut neutre autant au point de vue du langage que des terminologies (Farrar and Lewis (2005) dans (Wilcock, 2007)). La neutralité de cette ontologie permet d'y intégrer différents types d'information, dont des informations correspondantes à une grammaire HPSG (Wilcock, 2007).

1.2.2.4 Conclusion

La génération profonde va donc mettre en place les bases du document que le générateur va produire. Toutefois, l'ensemble des informations récupérées lors de cette première étape n'est pas toujours suffisant à la création du texte final. Pour reprendre l'exemple du système expert donné dans l'historique (Swartout, 1983), le fait d'avoir un thème sur lequel produire n'est pas le seul besoin du générateur. Il faut souvent lui fournir d'autres informations afin de compléter ses connaissances. Généralement, les informations complémentaires seront fournies lors de la génération de surface du document, le module ajouté ayant une connaissance du domaine.

1.2.3 La génération de surface

La génération profonde regroupe l'ensemble des informations pertinentes pour l'utilisateur dans une pseudostructure de document. Cette structure est inutilisable pour le lecteur et ne prend véritablement une allure de texte lisible qu'une fois passée à travers le générateur de surface.

Ce dernier va, dans les faits, effectuer un grand nombre d'opérations, dont notamment celle de répondre aux questions 'à qui dire ?' et 'comment dire ?'.

1.2.3.1 Le 'à qui dire ?'

Cette question est notamment pertinente dans le domaine des tutoriels intelligents et de l'*e-learning* où le texte à produire doit être d'un niveau de langue similaire à celui de l'utilisateur et adapté à ce dernier. Le texte produit ne sera pas le même en fonction du pourquoi de la production ou de la personne qui effectue la demande. Le 'à qui dire' et le 'pourquoi dire' sont donc, à mon sens, étroitement liés

Le cas du 'pourquoi effectuer la génération' n'est pas expliqué dans la littérature et va dépendre, à mon sens, du but du générateur lui-même. Dans la plupart des cas, il s'agit de générer des phrases ou des textes afin de vérifier leur validité linguistique. Dans ce cas, le but de la génération est trivial et ne modifie en rien la façon de générer du logiciel. Toutefois, dans le cas des systèmes tutoriels intelligents (S.T.I.) ou d'un logiciel d'*e-learning*, la réponse à la question pourquoi générer prend une autre dimension, car le but de la génération n'est plus alors de transmettre une information, mais de transmettre un savoir. Il s'agit donc pour le logiciel d'être capable de planifier cette transmission des connaissances à fournir afin de ne pas perdre l'utilisateur et de vérifier que ce dernier comprenne ce qui lui est fourni. On passe alors dans une situation de communication où la génération est à la base du dialogue homme-machine.

Dans cette situation de communication, le programme doit considérer chaque utilisateur comme des entités distinctes ayant chacune leur propre vécu et donc leurs propres connaissances. De par ce fait, le texte fourni doit être pertinent et la redondance la moins présente possible. Cela signifie que le programme doit non seulement gérer les connaissances à fournir, mais également celles qu'il a précédemment fournies. Par précédemment, on sous-entend que le programme doit non seulement connaître les savoirs fournis plus tôt dans la séance d'enseignement, mais également celles développées lors des cours précédents (dans le cas des systèmes tutoriels intelligents - S.T.I. - notamment) (Woolf, 2009).

Malgré tout, cette question reste également pertinente dans le domaine de la Linguistique où elle aborde les théories de l'énonciation et leurs formalisations (Culioli, 1999).

1.2.3.2 Le 'comment dire ?'

La recherche du 'comment dire' ou du 'comment formuler' les idées hiérarchisées lors de la génération profonde est une recherche sur la formalisation de la langue. Les principales interrogations qui résultent de cette recherche sont liées au lexique ainsi qu'à la syntaxe liée à un mot donné.

Pour exemple, l'utilisation de pronom oblige à la création de lien pour résoudre les ambiguïtés possibles. Mais sans aller chercher aussi loin, les cas de la concordance des temps entre les différents verbes des différentes phrases du texte, l'enchaînement des phrases et leurs divers liens, ou tout simplement le choix d'un vocabulaire adapté peuvent amener des problèmes, notamment au niveau des interactions syntaxe-lexique. Gérard Sabah dans son ouvrage (1989) cite le fait que le choix d'un mot peut modifier la structure syntaxique de la phrase, par exemple le cas des verbes 'acheter' et 'vendre' qui selon le choix d'utiliser l'un ou l'autre, modifie la forme syntaxique, le thème de la phrase, mais peut également modifier la voix de cette dernière.

C'est cette recherche sur la formalisation du contenu en un texte cohérent qui reste la principale préoccupation des chercheurs en G.A.T.. Cependant, de nombreux formalismes développés durant ces dernières années tendent à faire disparaître la double génération (génération profonde et génération de surface). C'est notamment le cas de la génération par grammaire d'arbres adjoints ou de la génération basée sur les HPSG. Ces deux types de grammaires possédant de fortes contraintes à la fois syntaxiques et lexicales, la production de la base est déjà lexicalement et syntaxiquement un texte lisible.

Il existe également d'autres types de générateur tels que les générateurs de texte basés sur des graphes. Cette génération est la descendance directe de la génération de texte préenregistré avec variables.

La réalisation d'un graphe est relativement simple. Après avoir défini plusieurs états, il suffit de créer des liens unissant chaque état. Un état peut être un mot, un bout de phrase ou une variable. Dans le cas de la variable, celle-ci peut également être remplacée par un mot ou un morceau de phrase, mais également par un autre arbre, ce qui permet l'élaboration de phrases plus complexes.

La figure 7 nous montre le chemin suivi par un générateur par graphe pour produire le texte suivant : *'Ton cousin lui au moins a fait de bonnes études ; au fait je t'ai dit que Mme Schlurp est morte ? Ce n'est pas surprenant avec tous ces punks qui trainent dans le quartier... Enfin comme je dis à la pharmacienne c'est la vie... Au fait, ma petite pitchounette serait bien mignone³² si elle nous préparait le café.³³* Le générateur connaît le point de départ et calcule le chemin à parcourir.

32 On notera que l'orthographe correcte est *mignonne*.

33 Génération réalisée le 15 mars 2010 par l'intermédiaire d'un générateur disponible sur le site de Charabia.net, *Viens voir Mamie* .

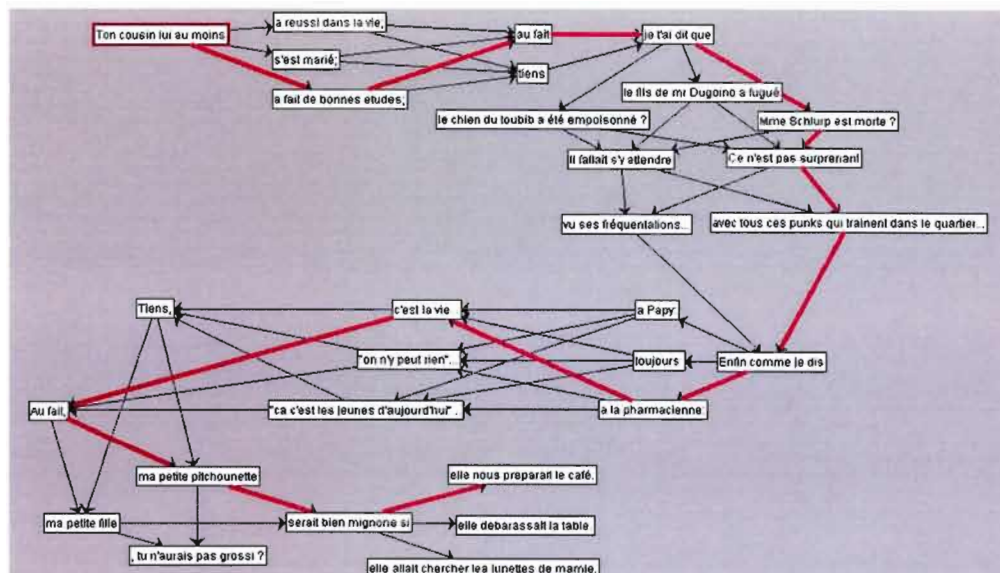


Figure 7 : Exemple de graphe utilisé dans la génération

Le générateur de cet exemple est un générateur de base, la plupart des générateurs par graphes étant beaucoup plus complexes dans le graphe et le calcul de la production. En général, il s'agit d'un calcul heuristique³⁴. Le choix de ce générateur se justifie par la facilité de compréhension de son fonctionnement. Toutefois, il s'agit ici d'une génération de surface sans génération profonde.

1.2.3.3 Conclusion

La réalisation de la structure de surface d'un texte, ou d'une phrase, est un processus extrêmement complexe. Les nouvelles avancées du domaine tendent de plus en plus à faire disparaître la génération de surface afin de simplifier le processus de génération.

³⁴ Calcul heuristique : calcul qui procède par approches successives en éliminant progressivement les alternatives et en ne conservant qu'une gamme restreinte de solutions tendant vers celle qui est optimale (CNRTL, *Centre National de Ressources Textuelles et Lexicales*).

Le fait de réaliser une structure de base (profonde) ayant déjà des liens sémantiques, lexicaux et syntaxiques permet de grandement faciliter la génération. Les ontologies sont d'ailleurs un excellent exemple de regroupement d'information pour la génération.

1.2.4 Conclusion

De plus en plus de recherches dans le domaine de la génération visent à regrouper la génération de surface et la génération profonde et cela pour plusieurs raisons. Tout d'abord le fait de devoir effectuer deux fois la génération peut devenir coûteux, au niveau du temps, en fonction des algorithmes choisis pour générer, mais également par le fait que le manque de connaissance dans la génération profonde implique que la structure du texte et des informations qu'il contient n'est souvent pas construite, mais donnée par le programmeur, ce qui limite les productions possibles (Sabah, 1989).

La génération profonde a également le désavantage d'être peu robuste et peu flexible quand elle est intégrée à des machines ayant des comportements intelligents (Sabah, 1989). Dans le cadre d'une recherche sur l'I.A., cela peut devenir problématique.

Cela a conduit les chercheurs à tenter de corriger les problèmes d'interactions entre les choix lexicaux, les choix syntaxiques et les décisions conceptuelles en mettant en place des grammaires formelles contraintes à la fois sur le lexique et la syntaxe.

1.3 Logiciels

Afin de réaliser mon projet, j'ai utilisé divers logiciels allant de l'étiqueteur syntaxique au logiciel permettant la création d'ontologies. Des explications sur ces divers logiciels sont données dans cette partie.

1.3.1 Tree-Tagger

Ce programme, développé par Helmut Schmid dans le cadre du projet Textcorpora à l'ICLUS (Institute for Computational Linguistics of the University of Stuttgart), effectue le marquage des mots de vocabulaire d'un texte avec divers types d'informations tels que le lemme ou la catégorie grammaticale du mot (nom, verbe, adjectif, etc.). Il marque également les éléments de ponctuation.

Ce programme permet d'effectuer plusieurs types d'étiquetage. Ces différents étiquetages se réalisent à divers niveaux. Tree-tagger peut pratiquer un découpage (chunk) sur un texte ou sur une phrase tout comme il peut appliquer des étiquettes morpho-syntaxiques sur des mots, mais il offre également la possibilité de lemmatiser automatiquement le texte. Le découpage consiste à étiqueter un texte en fonction des différentes parties du discours du texte. Il est toutefois envisageable de réaliser l'étiquetage et la lemmatisation d'un texte simultanément et de récupérer en sortie un fichier contenant sur chaque ligne un mot du texte, son étiquette syntaxique et son lemme.

L'un des avantages à utiliser le Tree-Tagger comme étiqueteur est que dans la majorité des cas, il sait désambiguïser comme le montrent les figures 8 et 9, même s'il lui arrive de laisser quelques erreurs comme on peut le constater dans la figure 9.

```
MacBook-Pro-de-Karl-Szymoniak:tree-tagger karlszymoniak$ echo 'il la porte la porte' | cm
d/tagger-chunker-french
reading parameters . .
tagging . .
finished.

<S>
<VN>
il      PRO:PER il
la      PRO:PER la
porte   VER:pres      porter
</VN>
<NP>
la      DET:ART le
porte   NOM          porte
</NP>
</S>
```

Figure 8 : Étiquetage des syntagmes de la phrase «Il la porte la porte.»

Comme on peut le constater sur la figure 9, il semblerait qu'il reste encore quelques bugs³⁵, car le logiciel admet des syntagmes nominaux ne contenant pas de nom, mais un adjectif à la place. Cette erreur est toutefois relativement rare. Cependant, une correction automatique de la présence d'un nom dans les NP a été effectuée. En outre, le logiciel commet d'autres types d'erreurs tels que la fermeture des syntagmes, également corrigés automatiquement. Afin de contrôler la validité des résultats, j'ai procédé à une relecture des patrons après les diverses corrections.

```
MacBook-Pro-de-Karl-Szymoniak:tree-tagger karlszymoniak$ echo il ferme d'une main ferme la porte de la ferme | cmd/tagger chunker-french
reading parameters ...
tagging ...
finished.

<s>
<VN>
il      PRO:PER 3s
ferme   VER:subp      ferme
</VN>
<PP>
d       PRP      de
<NP>
une     DET:ART un
main    NOM      main
</NP>
</PP>
<AP>
ferme   ADJ      ferme
</AP>
<NP>
la      DET:ART le
porte   NOM      porte
</NP>
<PP>
de      PRP      de
<NP>
la      DET:ART le
ferme   ADJ      ferme
</NP>
</PP>
</s>
```

Figure 9 : Étiquetage des syntagmes de : «Il ferme d'une main ferme la porte de la ferme»

Il aurait fallu que j'effectue quelques tests afin de connaître le pourcentage d'erreur de l'étiqueteur et du chunk du programme. Mais je pense au nombre d'erreurs que j'ai dû automatiser qu'il doit rester près de cinq pour cent d'erreurs au niveau du corpus.

³⁵ Erreur de programmation due au développeur et conduisant à un mauvais fonctionnement du logiciel. À l'origine, il s'agissait d'insectes qui, attirés par les ampoules des premiers ordinateurs, entraient dans la machine et y mouraient en causant des courts circuits.

1.3.2 L'ontologie par Protégé

1.3.2.1 Ontologie

Une ontologie est une méthode pour représenter un domaine de connaissance. Ce domaine va être capturé sous forme de concepts³⁶ dans l'ontologie, que l'on appelle les classes. Mais l'ontologie ne fait pas qu'énumérer les concepts d'un domaine, elle permet également d'explicitier les différentes relations qui interviennent entre les concepts du domaine. La majorité des ontologies lient les concepts qui les composent avec deux types de relations :

- une relation taxinomique
- une relation sémantique

L'ontologie va fournir, en plus des concepts et de leurs relations, une terminologie du domaine ainsi qu'une spécification de la signification des termes de ce vocabulaire. Elle permet également de créer une classification des connaissances. La plupart du temps, elle se fait sous la forme de schémas de données.

La majorité des langages utilisés pour écrire les ontologies est basée sur la logique de prédicat (Lepage, 2001) et permet de représenter les connaissances sous forme d'assertion ayant un individu, un prédicat et une classe.

Le W3C³⁷, dans le cadre de travaux effectués sur le Web sémantique³⁸ (W3C, 2001), a organisé des groupes de travail dédiés à l'implémentation de langages standard pour

³⁶ Construction de l'esprit explicitant un ensemble stable de caractères communs désignés par un signe verbal. Le concept regroupe les objets qu'il définit en une même catégorie appelée « classe » (*Dictionnaire de l'Académie Française*).

³⁷ World Wide Web Consortium.

³⁸ Il s'agit d'un effort collaboratif mené par le W3C avec la participation d'un grand nombre de chercheur et d'associés industriels. Le Web sémantique est basé sur des modèles de graphes qui représentent les ressources du Web dans un langage formel : le RDF (Resource Description Framework). À l'heure actuelle, ce langage a toutefois évolué pour être plus facilement compréhensible.

permettre le partage et la réutilisation des données. L'un de ces groupes était dévoué à la recherche d'un langage ontologique permettant l'utilisation et l'échangeabilité d'ontologies sur le Web. C'est en 2004 que ce groupe a publié ses recommandations définissant le langage OWL³⁹ comme le standard des langages ontologiques. Basé sur la technologie RDF⁴⁰, le langage OWL ajoute à cette dernière une syntaxe XML⁴¹.

La standardisation du format OWL par le W3C et l'intérêt qu'apporte cette représentation des connaissances, telle que Protégé la révèle, a amené plusieurs groupes de recherche à travailler de plus en plus avec ce type de représentation. Ce qui explique le succès du logiciel.

1.3.2.2 Protégé

Le programme utilisé afin de construire mon ontologie syntaxique s'appelle Protégé. Il s'agit de la quatrième version de ce logiciel développé par l'Université de Stanford dans le cadre de leur faculté de médecine. Ce logiciel fut destiné en tout premier lieu à la création d'une représentation des connaissances médicales dans le but de réaliser un système tutoriel intelligent destiné à l'enseignement et aux dépistages des différents symptômes et maladies liées chez les patients pour les étudiants en médecine.

Comme je l'ai expliqué précédemment, la standardisation du format OWL et la façon de représenter les informations à l'aide de ce logiciel ont mené au développement de nombreuses ontologies dans des domaines totalement différents du domaine d'origine du logiciel.

³⁹ Web Ontology Language.

⁴⁰ Resource Description Framework.

⁴¹ eXtensible Markup Language.

Afin d'en expliquer le fonctionnement, je vais reprendre les exemples fournis dans la documentation (p 10 à 12 de (Horridge, 2009)) sur la construction d'une ontologie à l'aide de Protégé.

Une ontologie n'est, à la base, composée que d'individus (ou instances). Ces individus représentent des éléments du domaine à implémenter (Figure 10).

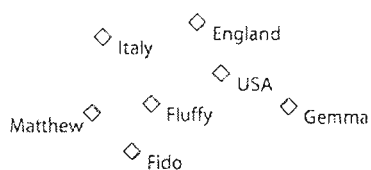


Figure 10 : Exemples d'individus présents dans une ontologie.

Mais une ontologie ne se limite pas seulement à contenir un certain nombre d'individus. Elle permet effectivement de lier les individus entre eux par des propriétés. Ces dernières sont élaborées par l'utilisateur en fonction de ses besoins. Les propriétés sont des relations binaires liant deux et seulement deux individus. Cela n'empêche en rien de lier un individu à plusieurs autres. Dans la figure 11, on voit l'apparition des liens entre Matthew et sa soeur Gemma, mais également entre Matthew et le pays dans lequel il habite.

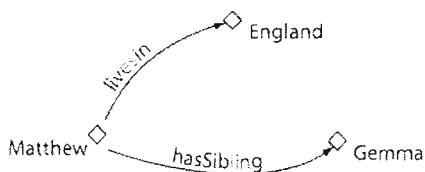


Figure 11 : Exemples de propriétés entre les individus d'une ontologie.

La configuration des ontologies au format OWL permet de créer des liens réversibles. Ainsi le lien 'hasSibling' pourrait être considéré comme une métapropriété comportant la propriété 'isTheBrother' (lien Matthew vers Gemma) ainsi que la propriété 'isTheSister' (lien Gemma vers Matthew).

Dans une autre approche des propriétés, on constate que le logiciel Protégé a la particularité d'accepter qu'une même entité puisse avoir plusieurs appellations. Pour reprendre un autre exemple de Matthew Horridge (2009), l'utilisation de l'individu 'la Reine Elizabeth', de 'The Queen' ou d'"Elizabeth Windsor" réfère toujours à la même entité. Une ontologie OWL classique ne permet pas d'associer la même entité à des noms d'entités différents. Le programme Protégé a cette particularité de faire lui même le rapprochement entre les individus en fonction des propriétés qui les spécifient. Prenons comme autre exemple l'entité Jean de la figure 12. On peut supposer que Jean n'a qu'une seule mère biologique. Si l'on crée vers deux entités différentes une même propriété, le programme va automatiquement lier ces deux entités comme étant une seule et même entité.

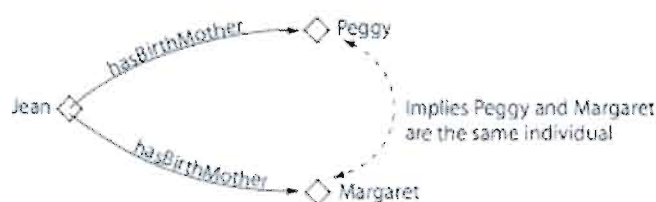


Figure 12 : Cas d'individus identiques, mais ayant un nom différent.

Néanmoins, si les deux entités, ici 'Peggy' et 'Margaret', ont explicitement été notées comme des entités différentes, alors le programme relèvera une incohérence dans la définition des propriétés mises en place.

Le logiciel Protégé permet également de créer des liens de transitivité (figure 13), de symétrie et de non-symétrie (figure 14) ainsi que des propriétés réflexives⁴² ou non réflexives (figure 15).

⁴² Relation mathématique R signifiant que dans un espace E, tout élément x appartenant à E se trouve lié à lui même par R.

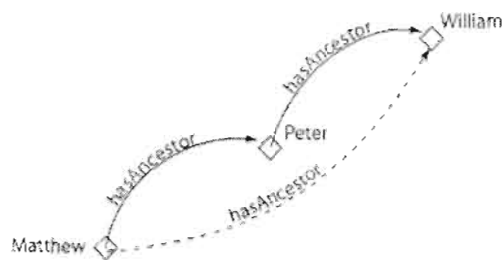


Figure 13 : Propriété transitive.



Figure 14 : Propriété symétrique et asymétrique.



Figure 15 : Propriété réflexive et non réflexive.

Mais les ontologies et le logiciel Protégé ne permettent pas uniquement de lister des individus et les propriétés qui les relient. Elles possèdent, afin de faciliter le traitement, la possibilité de regrouper les individus dans des classes (figure 16). Ces classes sont la représentation des concepts de l'ontologie.

Les propriétés définies par le concepteur de l'ontologie peuvent alors se produire entre entités de mêmes classes ou de classes différentes. Le logiciel permet également de créer des propriétés qui s'appliquent d'un individu vers une classe entière ou entre deux classes.

Pour exemple, un individu allergique aux animaux se verra attribuer une propriété s'appliquant non pas à un type d'animal, mais à l'ensemble de la classe regroupant les animaux.

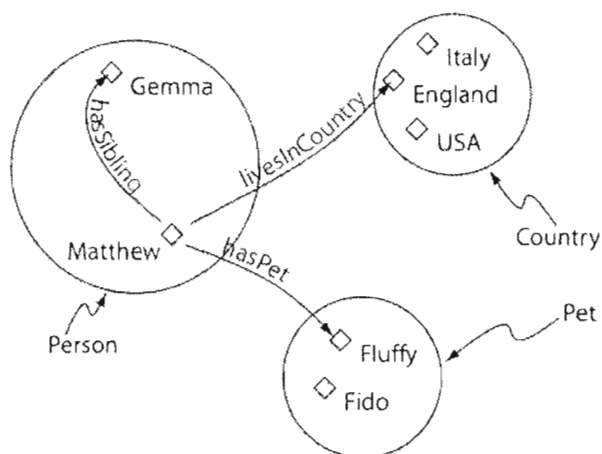


Figure 16 : Exemple de classes

1.3.3 LKB

Le LKB (Linguistic Knowledge Builder) est un logiciel permettant d'implémenter des grammaires de type HPSG. Ce programme a été réalisé dans les laboratoires de recherche en Informatique de l'Université de Cambridge dans le cadre du projet ACQUILEX⁴³.

Ce programme permet de générer un lexique marqué des principaux traits des grammaires de types HPSG. Pour cela, l'utilisateur construit sa propre grammaire en fonction de ce qu'il veut obtenir. Le LKB a pour avantage de pouvoir intégrer des grammaires très simples, dans lesquelles peu de liens sémantiques (voire aucun) apparaissent tels que dans la

⁴³ ACQUILEX est un projet de recherche lancé par l'Union Européenne sur la construction d'une base de connaissances lexicales multilingues pour faciliter l'implémentation d'une machine capable de traduire dans diverses langues un texte d'une langue donnée.

figure 17, où l'on se trouve en présence d'une grammaire de base où seul l'accord en nombre a été ajouté. La figure 17 contient également l'ensemble des informations nécessaires à la génération de la structure de traits typés, telles que les types de la structure syntaxique (explicités sous : `;;; Types`), le lexique (`;;; Lexicon`) ainsi que les règles de la grammaire (`;;; Rules`).

Toutefois, un problème dû à l'ancienneté de la version du LKB, dont la dernière version stable date de 1997, fait que la génération de la grammaire ne fonctionne pas. Pensant à un problème de version avec l'OS de Microsoft Windows 7, j'ai essayé le LKB sur diverses plateformes antérieures : Vista, XP (SP1, 2 et 3) ainsi que sur Millénium. Je n'ai pas pu remonter à Windows 98, plateforme originelle du LKB. Malgré les informations fournies sur le site, je n'ai pas été en mesure de faire fonctionner le générateur et j'ai donc dû rechercher une autre méthode pour marquer mon vocabulaire.

```

;;;Types
syn-struct = *sup* &
[ CATEG "cat."
  NUMAGR "agr" ]

agr = *sup*
eg = *eg*

pl = *agr*

phrase = *syn-struct* &
[ ARGS "list" ]

word = *syn-struct* &
[ ORTH "string" ]

eg-word = word &
[ NUMAGR "eg" ]

pl-word = word &
[ NUMAGR "pl" ]

;; Start structure
start = phrase &
[ CATEG "s" ]

;; Lexicon
this = eg-word &
[ ORTH "this"
  CATEG "det" ]

there = pl-word &
[ ORTH "there"
  CATEG "det" ]

the = word &
[ ORTH "the"
  CATEG "det" ]

;;; LEXICON CONTINUED

sleep = pl-word &
[ ORTH "sleep"
  CATEG "v" ]

sleeps = eg-word &
[ ORTH "sleeps"
  CATEG "v" ]

dog = eg-word &
[ ORTH "dog"
  CATEG "n" ]

dogs = pl-word &
[ ORTH "dogs"
  CATEG "n" ]

;; Rules
a-rule = phrase &
[ CATEG "s"
  NUMAGR #1
  ARGS [ FIRST [ CATEG "sp"
    NUMAGR #1 ]
    NEXT [ FIRST [ CATEG "v"
    NUMAGR #1 ]
    NEXT "null" ] ] ]

np-rule = phrase &
[ CATEG "np"
  NUMAGR #1
  ARGS [ FIRST [ CATEG "det"
    NUMAGR #1 ]
    NEXT [ FIRST [ CATEG "n"
    NUMAGR #1 ]
    NEXT "null" ] ] ]

```

Figure 17 : Exemple de grammaire à produire pour la réalisation d'une structure de traits.

1.5 Conclusion

Bien que des linguistes aient participé à l'élaboration de certains logiciels (LKB, Tree-tagger) le champ de la recherche dans la G.A.T. reste majoritairement aux mains des informaticiens, et c'est dans ce cadre que s'intègre mon projet de maîtrise.

Effectivement, la production écrite est un acte langagier dont l'étude fait partie de la linguistique (Culioli, 1999). Le fait de faire produire par un programme une phrase ou un texte, ayant des caractéristiques comparables à une production humaine, entre également dans le champ de la linguistique et devrait être produit en véritable collaboration entre informaticiens et linguistes.

L'implémentation d'un algorithme peut être du domaine de l'informatique, mais la programmation de la structure d'une langue doit être effectuée par des personnes ayant les connaissances de cette structure.

Le but de mon projet s'insère dans ce cadre où le linguiste n'intervient que trop peu souvent. Il va s'agir de créer un programme qui sera capable de générer des phrases acceptables et comparables à des productions humaines, sans toutefois avoir la prétention de pouvoir passer le test de Turing. Il s'agit donc de générer des phrases syntaxiquement et sémantiquement correctes.

Pour ce faire, je vais développer une ontologie syntaxique afin de montrer que les connaissances ontologiques peuvent permettre la génération de phrases syntaxiquement correctes. Les liens sémantique de l'ontologie appuyés par le lexique marqué à l'aide des structures de traits produit par du LKB devraient permettre une production sémantique.

CHAPITRE II

MÉTHODOLOGIE

Afin de pouvoir réaliser mon ontologie syntaxique et mon programme de génération, il m'a fallu trouver une base à partir de laquelle je puisse extraire un ensemble de patrons syntaxiques et ainsi créer la base de ma future ontologie.

J'ai choisi de travailler à partir d'un corpus d'articles journalistiques tirés des publications du journal 'Le Monde'. Le format de mon corpus n'étant pas exploitable sous sa forme de base, j'ai réalisé divers programmes de mise en forme pour arriver au corpus voulu, tout cela dans le but de faciliter le traitement des données pour l'ontologie.

Toutefois, avant d'en arriver à un corpus utilisable, de nombreuses manipulations ont été nécessaires, comme le montre la figure 18. Cette figure montre la méthodologie telle qu'elle était prévue au début de mon projet. Une modification a dû être apportée au schéma afin de le faire coïncider avec les étapes réelles de la réalisation de mon projet. Ce nouveau schéma est montré en figure 19.

La première étape fut donc de mettre en forme le corpus à l'aide d'un script en Perl (1) afin de recueillir l'ensemble des lexèmes du corpus. Une fois cette étape effectuée, j'ai marqué catégoriellement l'ensemble des lexèmes avec le logiciel Tree-Tagger (2).

Après avoir récupéré mon vocabulaire catégorisé, j'en ai extrait les patrons syntagmatiques⁴⁴ (s, NP, PP, VN, etc.) avec lesquels j'ai ensuite créé mon ontologie (3).

⁴⁴ La terminologie utilisée pour le marquage des patrons est celle fournie par le Tree-tagger.

Par la suite, une demande utilisateur à mon programme (4) commande le questionnement (5) de l'ontologie et du vocabulaire catégorisé afin de produire une phrase syntaxiquement et sémantiquement correcte (6).

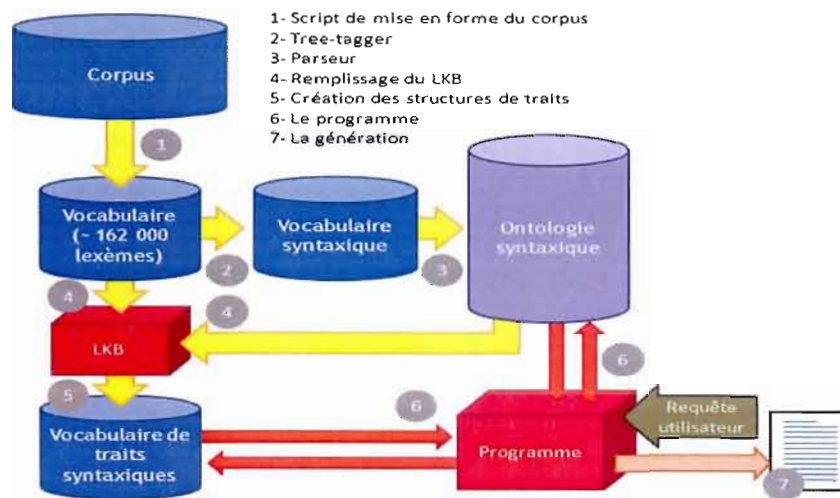


Figure 18 : Organigramme de la méthodologie originelle.

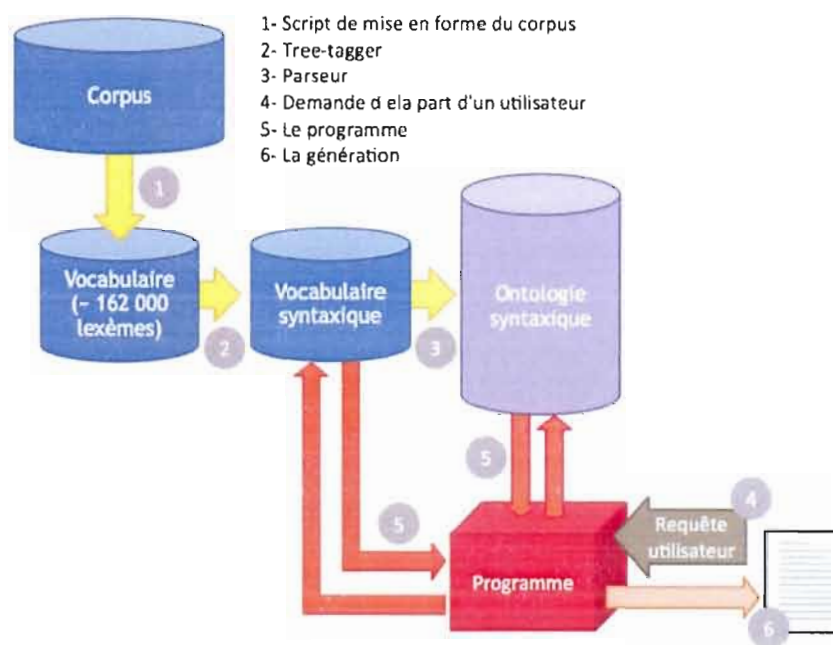


Figure 19 : Nouvel organigramme de la méthodologie.

2.1 Corpus

Les données de mon corpus proviennent du site internet du journal 'Le Monde'. La récupération des données sur le web a été effectuée à l'aide de divers outils de veille, téléchargeant l'ensemble des flux RSS⁴⁵ du journal entre le 20 novembre 2006 et le 20 mars 2008. Les outils utilisés pour le recueil de ces données me sont inconnus, le corpus m'ayant été fourni par M. Serge Fleury, l'un de mes professeurs au cours de ma première année de Master en Traitement Automatique de la Langue à l'Université de Paris 3 - La Sorbonne Nouvelle.

Le corpus est contenu dans un répertoire du même nom et dont l'arborescence est partiellement montrée à la figure 20.

On peut constater que les données du corpus sont classées dans une arborescence contenant quatre dossiers imbriqués. Chaque fichier est donc classé selon cinq critères. Les quatre premiers représentent, dans l'ordre, l'année, le mois, le jour et l'heure de récupération et correspondent à chaque dossier et sous-dossier. Vient ensuite le cinquième critère qui correspond à la rubrique de l'article. Pour exemple, dans la figure 20, 3208 correspond à la rubrique « À la Une », 3210 correspond à la rubrique « Internationale ».

On remarque également que chaque rubrique d'article existe sous deux types de fichiers :

- un fichier XML. Ce fichier comporte les informations sur la structure de l'information dans l'article : emplacement du titre, du texte, des liens publicitaires (figure 21).
- un fichier texte (txt). Ce fichier reprend le titre de l'article et le texte de ce dernier. Il comprend également diverses informations telles que la date et le numéro de l'article (figure 22).

⁴⁵ Un flux RSS désigne un flux de données encodé à l'aide d'une famille de formats XML (eXtensible Markup Language) utilisés pour la syndication (la vente de droit de reproduction et de diffusion de contenus) d'informations sur le Web (Wikipedia, *Flux RSS*)

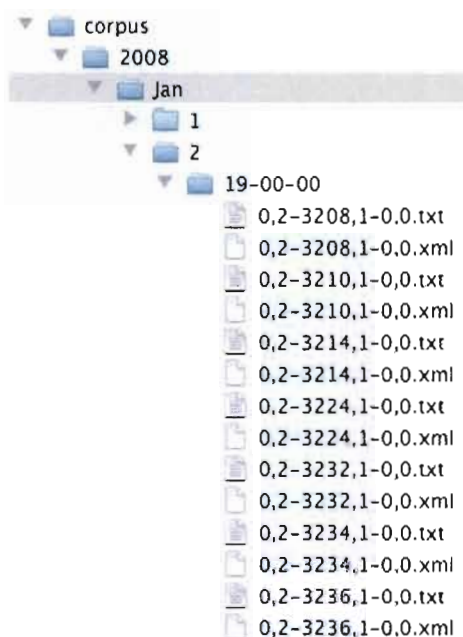


Figure 20 : Arborescence du répertoire « corpus ».

```
<?xml version="1.0" encoding="iso-8859-1"?><rss version="2.0" xmlns:rd="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<channel>
<title>Le Monde.fr : A la une</title>
<link>http://www.lemonde.fr</link>
<description>Toute l'actualité au moment de la connexion</description>
<copyright>Copyright Le Monde.fr</copyright>
<image><url>http://medias.lemonde.fr/mmpub/img/lgo/lemondefr_rss.gif</url><title>Le Monde.fr</title><link>http://
www.lemonde.fr</link></image>
<pubDate>Tue, 01 Jan 2008 17:58:07 GMT</pubDate></item>
<title>Un document révèle l'étendue des fraudes électorales au Kenya</title>
<link>http://www.lemonde.fr/web/article/0,1-0@2-3212,36-994926,0.html?xtor=RSS-3208</link>
<description>Un document confidentiel, que s.&#38;#39;est procuré &#38;#34;Le Monde&#38;#34;, trace les contours du
désastre entraîné par le scrutin du 27 décembre, qui vient de plonger le Kenya dans une vague de violences.</
description>
<pubDate>Tue, 01 Jan 2008 15:06:05 GMT</pubDate>
<guid isPermaLink="false">http://www.lemonde.fr/web/article/0,1-0@2-3212,36-994926,0.html?xtor=RSS-3208</guid>
<enclosure url="http://medias.lemonde.fr/mmpub/ed/ill/2008/01/01/h_1_ill_994932_keny2.jpg" type="image/jpeg"
length="2214"></enclosure>
</item>
<item>
<title>Mitt Romney et Mike Huckabee, les deux visages du Parti républicain</title>
<link>http://www.lemonde.fr/web/article/0,1-0@2-3222,36-994961,0.html?xtor=RSS-3208</link>
<description>La guerre pour l&#38;#39;investiture républicaine dans l&#38;#39;Iowa, dont le caucus du 3 janvier est capital,
fait rage entre les deux hommes. M. Huckabee, qui a effectué une percée inattendue, est en tête selon les derniers
sondages.</description>
<pubDate>Tue, 01 Jan 2008 15:15:19 GMT</pubDate>
<guid isPermaLink="false">http://www.lemonde.fr/web/article/0,1-0@2-3222,36-994961,0.html?xtor=RSS-3208</guid>
<enclosure url="http://medias.lemonde.fr/mmpub/ed/ill/2008/01/01/h_1_ill_995131_rom.jpg" type="image/jpeg"
length="2030"></enclosure>
</item>
```

Figure 21 : En-tête et premières lignes d'un fichier XML.

```

<FILE-date="2008/01/01/19">
<article-nb="2008/01/01/19-1">
<filnamedate="20080101"><AAMM="200801"><AAMMJJ="20080101"><AAMMJJHH="2008010119">
<filename="SURF-0,2-3208,1-0,0-1"> Un document confidentiel, que s&#38;#39;est procuré &#38;#34;Le Monde&#38;#34;,
    trace les contours du désastre entraîné par le scrutin du 27 décembre, qui vient de plonger le Kenya dans une vague de
    violences.
<filename="PROF-0,2-3208,1-0,0-1">
    Un document confidentiel de sept pages, que s'est procuré Le Monde,
    trace les contours du désastre entraîné par le scrutin du 27 décembre
    2007, qui vient de plonger le Kenya dans une vague de violences. Le
    texte reconse les irrégularités constatées par les observateurs des
    partis. Il a été annoté par un participant lors d'une réunion discrète
    rassemblant des partis en lice. Cette réunion s'est tenue au cours de
    la nuit du samedi 29 au dimanche 30 décembre, celle précédant l'annonce
    des résultats de la présidentielle.

```

Au moins 15 personnes brûlées vives dans un
église dans l'ouest du Kenya

L'incendie criminel d'une église dans laquelle s'étaient réfugiés
plusieurs centaines de personnes cherchant à fuir les violences
post-électorales a fait entre quinze et trente morts à Eldoret, mardi
1er janvier, selon les sources. "Une foule énorme a attaqué l'église",

Figure 22 : En-tête et premières lignes d'un fichier texte.

Les fichiers qui seront traités pour remplir l'ontologie et récupérer le vocabulaire seront les fichiers textes. Comme le montre la figure 22, on peut constater le problème de mise en page du document où la première lettre du premier mot de la première ligne de l'article se trouve être séparée du reste du mot par un espace et ce phénomène se répète pour chaque début d'article. Toutefois le problème ne se pose pas si la première lettre est une lettre-mot telle que la préposition À⁴⁶. Mais je reviendrai plus longuement sur ce problème dans la section 3.1.1 du chapitre 3.

2.2 Langage

Le langage utilisé pour le développement de mon programme est le langage Perl. Il a été développé par Larry Wall en 1987 (Wall, Christiansen et Orwant, 2001). J'ai choisi

⁴⁶ Notons qu'un autre problème est soulevé dans ce cas, celui de l'accent. L'écriture des majuscules accentuées, obligatoire au Québec, n'est nullement une obligation en France, bien que de plus en plus utilisée.

d'implémenter mon programme avec ce langage pour plusieurs raisons, la principale étant que Perl a été élaboré dans le but de faciliter la manipulation de données textuelles.

Perl est un langage script qui a l'avantage d'être relativement accessible et intuitif dans son implémentation, surtout pour une personne n'étant pas programmeur. Ces avantages ne sont pas fortuits, le développement ayant été effectué pour faciliter la programmation par les linguistes (Tanguy et Hathout, 2007).

Une autre des raisons qui m'a incité à utiliser Perl, est qu'il existe, pour ce langage script, un très grand nombre de modules (CPAN)⁴⁷ que l'on peut rajouter à la version initiale dans le but de gérer les divers types de données et de pouvoir travailler avec plusieurs programmes différents. Ces modules apportent une souplesse de programmation qui ne se retrouve pas dans d'autres langages utilisés plus couramment.

Parmi ces modules, on en retrouve permettant la création d'une interface qui se présente sous la forme de fenêtres de dialogue qui rend les «échanges» personne-machine moins austères que la voie classique de la gestion des scripts à l'aide de la console du terminal⁴⁸.

Pour finir, Perl est un langage que j'apprécie et que j'utilise régulièrement. Je possède donc une base de programmation solide dans ce langage, base que je ne possède pas dans d'autres langages tels que Java ou Cocoa.

2.3 Sous-programmes

Dans le cadre de la réalisation de mon projet, j'ai dû réaliser un ensemble de programmes annexes afin de mener à son terme la génération d'une phrase à l'aide d'une ontologie syntaxique. La plupart de ces programmes ne concernent pas directement le

⁴⁷ Comprehensive Perl Archive Network.

⁴⁸ Le dialogue à l'aide de la console du terminal ne s'effectue qu'à l'aide de lignes de code.

générateur et s'appliquent principalement à la mise en forme du corpus de base afin que le générateur puisse avoir accès à des données hiérarchisées et surtout utilisables.

Les programmes implémentés sont efficaces, même s'ils ne sont pas les plus performants. Ce défaut peut s'expliquer par le fait que n'étant pas programmeur, je ne connais pas les algorithmes les plus efficaces pour arriver à optimiser mes programmes.

Les programmes de traitement du corpus sont aux nombres de six et décrits dans la suite de ce chapitre.

Tous les programmes créés afin de traiter le corpus et de faciliter l'implémentation de mon générateur de texte se trouvent dans l'annexe deux de ce mémoire.

2.3.1 Programme de récupération des données dans le corpus

Ce programme va lancer une fenêtre de dialogue écrite en Perl à l'aide du module Tk, module d'interface graphique multiplateforme (figure 23 et 24).

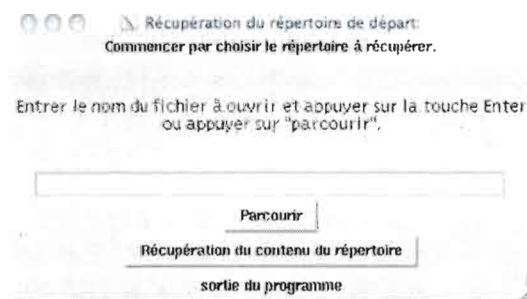


Figure 23 : Boîte de dialogue initiale.

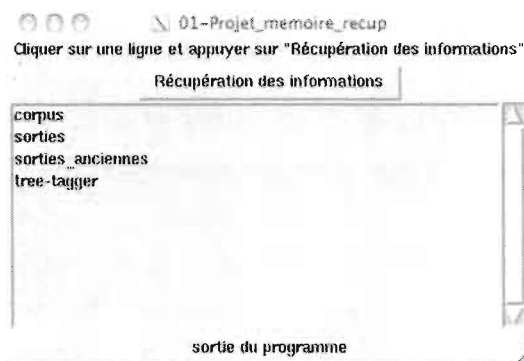


Figure 24 : Boîte de dialogue secondaire.

Le premier programme va demander à l'utilisateur le dossier dans lequel se trouve le corpus à traiter comme montré sur la figure 23. L'utilisateur a alors le choix de rentrer l'adresse du dossier et de cliquer sur récupération ou de cliquer sur parcourir. Cette seconde action va modifier la fenêtre (figure 24) en faisant apparaître l'ensemble des dossiers présents dans le dossier actuel. L'utilisateur n'a plus qu'à cliquer sur le nom du dossier contenant le corpus puis sur le bouton de récupération.

Une fois que le programme connaît le répertoire de départ, ici le dossier corpus, il va descendre dans l'arborescence du dossier et récupérer dans chaque fichier texte les informations concernant le titre et le contenu de l'article. Je n'effectue la recherche que dans les fichiers textes, les fichiers XML ne contenant qu'un rapide résumé des articles.

Ces informations recueillies dans chaque fichier vont être stockées dans un fichier XML ne comportant que deux balises :

- une balise titre
- une balise article

Par la suite, seule la balise article sera exploitée, la phrase de la balise titre étant, en général, reprise dans le texte de l'article. J'aurais pu faciliter ma programmation en ne récupérant que le contenu de l'article, mais je souhaitais également, au début de mon projet,

travailler sur la structure syntaxique des titres. J'ai abandonné cette idée par la suite, car je me suis retrouvé en présence d'un très grand nombre de phrases nominales que je ne pouvais pas utiliser par la suite.

La récupération du texte ne s'est pas faite sans heurts, le journal ayant semble-t-il changé l'encodage de ses données au cours de l'année 2007, bien que l'entête du flux RSS considère le contenu en CP1252⁴⁹. Le module d'affichage des pages dans le navigateur internet gère ces changements, mais un simple programme de lecture de fichier en Perl ne le fait malheureusement pas. La résolution du problème fut donc d'effectuer un remplacement de caractères, les problèmes d'encodages concernant, en général, les caractères accentués.

Après traitement de l'ensemble du corpus, ce sous-programme aura généré l'équivalent de trois cent sept mille deux cent quarante-deux lignes dans l'ensemble des fichiers créés pour un nombre de cinquante et un mille deux cent sept articles à travers toutes les rubriques.

2.3.2 Programme de création de phrases

Le but de ce programme est de découper les articles du fichier xml, généré par le premier sous-programme, en phrases. Ce sous-programme se nomme 02-Projet_memoire_phrases.pl.

Pour cela, j'ai effectué une recherche des signes de ponctuation de fin de phrases dans chaque ligne de chaque article. Un extrait d'un des fichiers de phrases est montré en figure 25.

Un des problèmes rencontrés est que certains mots contiennent des signes de ponctuation de fin de phrases sans qu'il s'agisse pour autant de la fin de la phrase. Pour exemple le mot monsieur se retrouve souvent dans le texte sous la forme M.. Il a donc fallu

⁴⁹ Encodage des caractères utilisé par défaut dans l'OS (Operating System) de Microsoft.

que je récupère l'ensemble des signes de ponctuation et que je fasse des vérifications sur les mots précédents et sur les mots suivants.

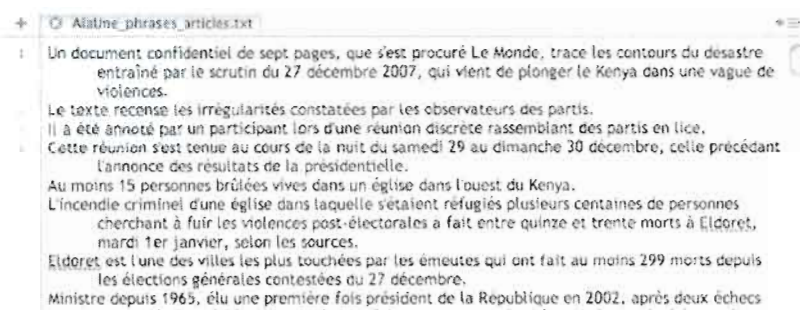


Figure 25 : Extrait d'un fichier de phrases.

Ainsi les mots d'une seule lettre suivie d'un point ne sont pas considérés comme des fins de phrases. Pour exemple, certaines références à des personnalités (auteur, homme politique, etc.) s'effectuent à l'aide de l'initiale du prénom de la personne suivie d'un point puis du nom de famille. Dans ce cas, le nom de famille ne doit pas se retrouver seul à la ligne. J'ai donc remplacé les initiales par le nom monsieur afin d'avoir un accès au nom de famille de la personne.

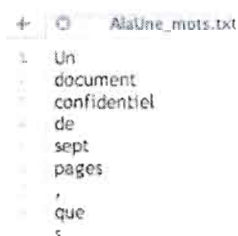
Malgré le marquage d'un grand nombre d'exceptions dans l'implémentation de mon code, certaines césures n'ont pas lieu au bon endroit. Une relecture des fichiers de phrases a donc été nécessaire afin d'éviter pour la suite les phrases ayant des patrons incomplets ou mal formés.

Le nombre de phrases obtenues, pour l'ensemble de mes fichiers à la fin de l'utilisation de ce script, est de six cent trente-trois mille sept cent cinquante-sept.

2.3.3 Programme de mise en mots

Une fois effectuée la création de chaque fichier de phrases, une nouvelle segmentation a été faite. Ce nouveau découpage fut nécessaire afin d'obtenir un mot par ligne dans chaque fichier. Pour mon programme, il ne s'agit pas seulement d'un mot de vocabulaire par ligne, mais également de toutes marques de ponctuation.

Pour exemple, une phrase comme : «Un document confidentiel de sept pages, que s'est procuré Le Monde, trace les contours du désastre entraîné par le scrutin du 27 décembre 2007, qui vient de plonger le Kenya dans une vague de violences.» sera découpée sur quarante et une lignes : «Un/document/confidentiel/de/sept/pages/,/que/s/'/est/procuré/Le/Monde/,/trace/les/contours/du/désastre/entraîné/par/le/scrutin/du/27/décembre/2007/,/qui/vient/de/plonger/le/Kenya/dans/une/vague/de/violences/.». Ce découpage est obligatoire pour l'étiqueteur utilisé. Un extrait est montré en figure 26.



```

+ AlaUne_mots.txt
1 Un
2 document
3 confidentiel
4 de
5 sept
6 pages
7 ,
8 que
9 s

```

Figure 26 : Extrait d'un fichier de mots.

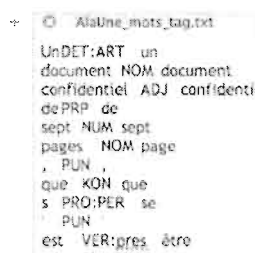
La mise à la ligne des ponctuations est nécessaire à l'étiqueteur. Toutefois, il existe une autre méthode avec le Tree-tagger. Cette alternative consiste à utiliser le chunker⁵⁰ du Tree-tagger. Dans ce cas, on entre directement la phrase et le programme la découpe lui-même automatiquement. Je reviendrai sur cette méthode au point 2.3.6.

⁵⁰ Le chunker est un programme d'étiquetage des parties du discours et permettant la délimitation des groupes syntagmatiques.

2.3.4 Programme d'étiquetage

L'étiqueteur utilisé pour annoter mon corpus est l'étiqueteur Tree-Tagger⁵¹. Un extrait de fichiers taggés est montré sur la figure 27.

La liste des catégories grammaticales d'origine utilisées par le parseur est en annexe quatre. Toutefois, j'ai effectué certaines modifications dans les catégories données par le logiciel.



```

UnDET:ART un
document NOM document
confidentiel ADJ confidentiel
de PRP de
sept NUM sept
pages NOM page
, PUN ,
que KON que
s PRO:PER se
PUN
est VER:pres être

```

Figure 27 : mots taggés.

Ces modifications concernent :

- l : que le programme considère comme un nom (NOM) et qui est un déterminant article⁵² (DET:ART).
- c : représentant le «ce» et que le programme annote comme un numéraire (NUM) et dont j'ai changé le marquage en pronom démonstratif (PRO:dem). Ce marquage étant celui utilisé pour toutes les autres occurrences du lemme «ce».
- s : provenant des verbes pronominaux et que le logiciel étiquette comme des verbes (VER) ou des adjectifs (ADJ), et que j'ai modifié en pronom personnel (PRO:PER), catégorisation de l'étiqueteur utilisé pour les «se» pronominaux.

⁵¹ Pour de plus amples explication sur le Tree-Tagger, se référer à la section: i.3.1.

⁵² Les catégories utilisées pour corriger l'étiquetage du Tree-tagger ont été vérifiées sur le site du Centre National de Ressources Textuelles et Lexicales (CNRTL, *Centre National de Ressources Textuelles et Lexicales*).

- j : le pronom personnel que le programme considère comme une abréviation (ABR) et que j'ai rétabli dans son rôle de pronom (PRO:per)
- qu : qui est annoté comme un nom (NOM). Je l'ai déplacé sous l'étiquette des conjonctions (KON), étiquette que le programme utilise pour marquer le lemme «que».
- n : de la négation que le programme considère comme un adjectif. J'ai changé sa valeur donnée par le programme par celle de la classe des adverbes (ADV).
- d : selon le cas, le «d» se trouve annoté comme un verbe (VER) ou un nom (NOM). En utilisant le marquage du lemme «de», j'ai réannoté le «d» comme une préposition à fonction de déterminant (PRP:det).

2.3.5 Programme de « re-création » des phrases

Ce programme récupère dans le fichier de mots taggés, les mots ainsi que leurs catégories grammaticales. En associant les deux, je modifie les phrases en y intégrant les catégories grammaticales (figure 28). La création des phrases est beaucoup plus simple à ce moment précis du processus, les phrases originelles ayant été corrigées plus tôt dans le processus. Les signes de ponctuation de fin de phrase n'ont donc que cette valeur possible.

J'agglutine les mots et leurs catégories sur une même ligne jusqu'à rencontrer une fin de phrase, catégorisée par SENT, comme montré sur la figure 28.

Un|DET:ART|document|NOM|confidentiel|ADJ|de|PRP|sept|NUM|pages|NOM|,|PUN|que|KON|s|
 PRO:PER'|PUN|est|AUX:pres|procuré|VER:ppre|Le|DET:ART|Monde|NAM|,|PUN|trace|
 VER:pres|les|DET:ART|contours|NOM|du|PRP:det|désastre|NOM|entraîné|VER:ppre|par|PRP|
 le|DET:ART|scrutin|NOM|du|PRP:det|27|NUM|décembre|NOM|2007|NUM|,|PUN|qui|PRO:REL|
 vient|VER:pres|de|PRP|plonger|VER:infi|le|DET:ART|Kenya|NOM|dans|PRP|une|DET:ART|
 vague|NOM|de|PRP|violences|NOM|.|SENT
 Le|DET:ART|texte|NOM|recense|VER:pres|les|DET:ART|irrégularités|NOM|constatées|VER:ppre|par|
 PRP|les|DET:ART|observateurs|NOM|des|PRP:det|partis|NOM|.|SENT
 il|PRO:PER|a|AUX:pres|été|AUX:ppre|annoté|VER:ppre|par|PRP|un|DET:ART|participant|NOM|lors|
 ADV|d|PRP:det|'une|DET:ART|réunion|NOM|discrète|ADJ|rassemblant|VER:ppre|des|
 PRP:det|partis|NOM|en|PRP|lice|NOM|.|SENT
 Cette|PRO:DEM|réunion|NOM|s|PRO:PER'|PUN|est|AUX:pres|tenue|VER:ppre|au|PRP:det|cours|NOM|
 de|PRP|la|DET:ART|nuit|NOM|du|PRP:det|samedi|NOM|29|NUM|au|PRP:det|dimanche|NOM|
 30|NUM|décembre|NOM|,|PUN|celle|PRO:DEM|précédant|VER:ppre|l|DET:ART|'PUN|annonce|
 NOM|des|PRP:det|résultats|NOM|de|PRP|la|DET:ART|présidentielle|ADJ|.|SENT
 Le|PRP:det|désastre|NOM|de|PRP|la|DET:ART|présidentielle|ADJ|.|SENT

Figure 28 : phrases taggées.

Je récupère également dans un autre fichier, tous les lemmes du vocabulaire. Je crée ainsi un fichier de phrases de lemmes comme montré sur la figure 29.

un document confidentiel de sept page , que se 'être procurer le Monde , tracer le contour du
 désastre entraîner par le scrutin du @card@ décembre @card@ , qui venir de plonger le Kenya
 dans un vague de violence .
 le texte recenser le irrégularité constater par le observateur du parti .
 il avoir être annoter par un participant lors de ' un réunion discret rassembler du parti en lice .
 ce réunion se ' être tenir au cour|cours de le nuit du samedi @card@ au dimanche @card@ décembre ,
 celui précéder le ' annonce du résultat de le présidentiel .
 au moins @card@ personne brûler vif dans un église dans le ' ouest du Kenya .
 Le ' incendie criminel de ' un église dans lequel se ' être réfugier plusieurs centaine de personne
 chercher à fuir le violence post - électoral avoir faire entre quinze et trente mort à Eldoret ,

Figure 29 : Phrases de lemmes.

2.3.6 Programme de découpage en syntagmes

Afin de réaliser le découpage des phrases recréées en syntagme, j'ai effectué diverses programmations afin que les syntagmes soient découpés correctement. Toutefois, certaines configurations m'ont posé problème dans la cession du texte.

Ce n'est qu'après de nombreuses tentatives de découpage que j'ai découvert que le logiciel Tree-Tagger pouvait effectuer cette segmentation. Comme expliqué dans le paragraphe 2.3.3, il s'avère que le programme du Tree-tagger permet une segmentation syntagmatique de la phrase ainsi qu'un étiquetage de chaque mot.

Cette méthode fut donc beaucoup plus rapide à réaliser que ma tentative de segmentation de phrase. Cependant, un certain nombre d'erreurs sont faites par le Tree-tagger lors de son exécution, notamment au niveau de la fermeture des syntagmes. Pour exemple un grand nombre de syntagmes prépositionnels ne contenait que la préposition.

Après l'automatisation de la correction de la majorité des erreurs de fermeture des syntagmes, j'ai comparé le lexique étiqueté lorsque le fichier d'entrée contenait un mot par ligne (cf : 2.3.4) avec celui étiqueté lors de l'utilisation du chunk.

Il s'avère que dans le deuxième cas, le marquage syntaxique et la lemmatisation sont effectués avec un nombre d'erreurs moindre. Cela provient du fait que le marquage s'effectue avec une connaissance du contexte et non plus juste sur un mot isolé. En effet, la plupart des corrections réalisées pour obtenir un vocabulaire correctement étiqueté en 2.3.4, sont automatiquement effectuées par le logiciel. Toutefois, d'autres types d'erreurs apparaissent (cf : figure 30⁵³) lors de ce traitement.

Comme le montre la figure 30, après le passage dans le chunk, le fichier de sortie ne possède qu'un élément par ligne : balise d'ouverture ou de fermeture de syntagme, ou un mot avec sa catégorie syntaxique et son lemme. Une «reconstruction» des syntagmes fut donc nécessaire afin d'avoir en sortie un syntagme par ligne tel que le montre la figure 31. Le chevron d'ouverture de phrase est également seul sur une ligne ainsi que tous les signes de ponctuation. La phrase de la Figure 31 correspond à la phrase utilisée pour l'exemple au point 2.3.3 (p.57) : «Un document confidentiel de sept pages, que s'est procuré Le Monde, trace les contours du désastre entraîné par le scrutin du 27 décembre 2007, qui vient de plonger le Kenya dans une vague de violences.».

⁵³ On remarquera que la figure 8 et la figure 31 sont identiques. En effet, afin d'éviter au lecteur de faire des aller-retours entre la page 33 et celle-ci, j'ai décidé de replacer dans cette section la figure 8, celle-ci illustrant parfaitement mes propos.

Le nombre d'erreurs étant moindre avec cette méthode, j'ai décidé de travailler avec le fichier de mots produit par le chunker, les erreurs restantes semblant moins nombreuses que lors du premier traitement et bien qu'ayant déjà corrigé une majorité des erreurs dues à l'étiquetage classique, je pense que la prise en compte du contexte a dû éliminer un nombre d'erreurs que je n'ai probablement pas vues sur les premiers fichiers.

Ce choix m'a tout de même posé quelques problèmes, car il rendait mes sous-programmes de mise en mots, d'étiquetage et de reconstruction des phrases obsolètes. En effet, à partir d'un fichier d'entrée à une phrase par ligne, je me retrouvais avec un fichier de sortie avec un mot ou un chevron par ligne. Je n'avais donc plus qu'à récupérer les mots pour recréer mes fichiers de mots catégorisés, les chevrons avec le lexique catégorisé correspondant au contenu des syntagmes (en plaçant un syntagme par ligne) et enfin reconstituer les phrases avec le vocabulaire catégorisé.

```
MacBook-Pro-de-Karl-Szymoniak:tree-tagger karlszymoniak$ echo "il ferme d'une main ferme
la porte de la ferme" |cmd/tagger-chunker-french
reading parameters ...
tagging ...
finished.

<S>
<VN>
il      PRO:PER il
ferme   VER:subp ferme
</VN>
<PP>
d       PRP      de
<NF>
une     DET:ART un
main    NOM      main
</NF>
<PP>
ferme   ADJ      ferme
</PP>
<NP>
la       DET:ART le
porte    NOM      porte
</NP>
<PP>
de       PRP      de
<NP>
la       DET:ART le
ferme    ADJ      ferme
</NP>
</PP>
</S>
```

Figure 30 : Étiquetage en syntagmes de : «Il ferme d'une main ferme la porte de la ferme».

```

<S>
<NP> DET:ART NOM <AP> ADJ </AP> </NP>
<PP> PRP <NP> NUM NOM </NP> </PP>
PUN
<Ssub> KON <VN> PRO:PER AUX:pres VER:pper </VN> <NP> DET:ART NOM </NP> </Ssub>
PUN
<VN> VER:pres </VN>
<NP> DET:ART NOM </NP>
<PP> PRP:det <NP> NOM </NP> </PP>
<VPpart> VER:pper </VPpart>
<PP> PRP <NP> DET:ART NOM </NP> </PP>
<PP> PRP:det <NP> NUM NOM NUM </NP> </PP>
PUN
<NP> PRO:REL </NP>
<VN> VER:pres </VN>
<VPinf> PRP <VN> VER:infi </VN> </VPinf>
<NP> DET:ART NOM </NP>
<PP> PRP <NP> DET:ART NOM </NP> </PP>
<PP> PRP <NP> NOM </NP> </PP>
</S>
<S>
<NP> DET:ART NOM </NP>
<VN> VER:pres </VN>

```

Figure 31 : Découpage d'un syntagme par ligne.

J'ai également reconstruit l'ensemble des phrases et des syntagmes sous forme de lemmes afin de garder tout de même une « trace » de la phrase.

Ce troisième sous-programme est probablement le plus long à se réaliser sur l'ensemble du corpus étant donné le fait qu'il doit découper les six cent trente-trois mille sept cent cinquante-sept phrases du corpus et en étiqueter chaque mot et symbole pour une création d'un total de sept millions quatre-vingt-seize mille sept cent vingt et un syntagmes et treize millions neuf cent quatre-vingts mille sept cent trente mots. Le traitement du corpus par ce seul script prend entre six et sept heures.

2.3.7 Conclusion

Après avoir implémenté la quasi-totalité de mes sous-programmes (six à la base), la programmation de la mise en syntagme à l'aide du Tree-tagger m'a fait changer de direction et supprimer les sous-programmes de :

- création d'un mot par ligne, le chunker effectuant déjà cette tâche,
- re-crédation des phrases avec le lexique taggé,

- la mise en syntagme des phrases taggées.

Le troisième sous-programme n'aura pas été entièrement supprimé. J'ai en effet récupéré une bonne partie du code implémenté pour le réinjecter dans le dernier sous-programme créé (03-Projet_memoire_syntagm_tag.pl). Ce sous-programme effectue à peu près les mêmes tâches que les trois anciens, mais dans un ordre différent.

L'ensemble des trois sous-programmes réalisés se trouve dans l'annexe 1.

2.4 Réalisation de l'ontologie

Malheureusement, bien qu'une grande quantité de modules soit disponible pour Perl permettant la transformation de fichier dans divers formats, aucun n'existe pour effectuer la réalisation d'une ontologie. Certains modules permettent la lecture d'ontologie, mais pas leur écriture. Il a donc fallu que je crée l'ontologie et son contenu «manuellement».

Dans ce but, il me fallait en premier créer une représentation de ce que je voulais décrire dans mon ontologie, cela signifie créer une représentation de la phrase, de ses syntagmes et du contenu de chaque syntagme (et «sous-syntagme»).

J'ai donc commencé par réaliser une représentation de la phrase en me basant sur le «Vade-Mecum» de la nouvelle grammaire (Maisonneuve, 2003) ainsi que sur la 32e édition du petit Grevisse (Grevisse, 2009).

Ces ouvrages classent les phrases en deux types distincts :

- les phrases simples
- les phrases complexes

Pour commencer, j'ai choisi d'intégrer dans mon ontologie les structures des phrases simples contenues dans mon corpus. En m'aidant des deux ouvrages cités ci-dessus, j'ai analysé la structure des phrases de mon corpus. Les phrases simples sont composées d'au

moins deux éléments qui ont la fonction de sujet et la fonction de prédicat. On peut également trouver un troisième type d'élément qui a la fonction de complément. Si l'on regarde ce qui compose chacune des fonctions, on fait alors émerger une structure syntaxique de base pour la phrase simple. En effet, le sujet est généralement composé d'un syntagme nominal (SN) composé d'un déterminant et d'un nom. De même, le prédicat est réalisé à l'aide d'un syntagme verbal (SV) dont la base est un verbe intransitif ou transitif, dans ce dernier cas, le SV contient également un complément du verbe. J'ai implémenté dans mon ontologie ces trois fonctions principales ainsi que les diverses catégories des constituants de ces fonctions (Figure 32, on retrouve en rouge les fonctions et en bleu les catégories des principaux constituants). Puis à l'aide de mes propriétés, j'ai recréé la hiérarchie qui unit tous ces éléments dans la structure de la phrase.

Afin de ne pas alourdir le schéma, je n'y ai pas intégré les propriétés que relient les divers éléments. Une partie des propriétés seront montrés un peu plus loin dans le mémoire afin de permettre la compréhension du fonctionnement de l'ontologie ainsi que sa structure interne.

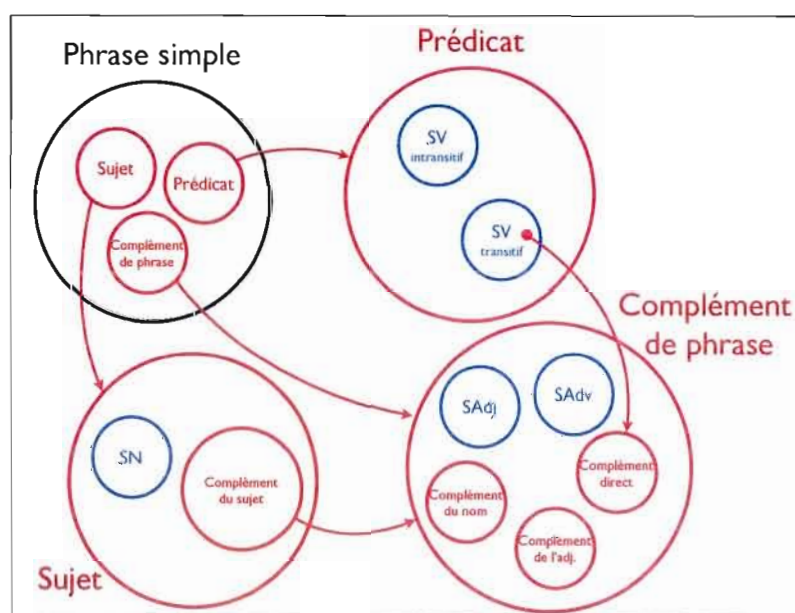


Figure 32 : Premier niveau de l'ontologie.

J'ai donc défini dans l'ontologie chacune des classes dont j'avais besoin pour construire mon générateur. Comme le montre la figure 32, les métaclasses sont représentées par les fonctions des constituants. Les propriétés ont été définies en fonction des « rôles » des divers constituants. Pour exemple : la propriété liant le nom au déterminant a été définie comme : 'est_Determine_par', la propriété inverse étant : 'Determine' (Figure 33). Les lignes jaunes dans la fenêtre de droite représentent les inférences du raisonneur⁵⁴.

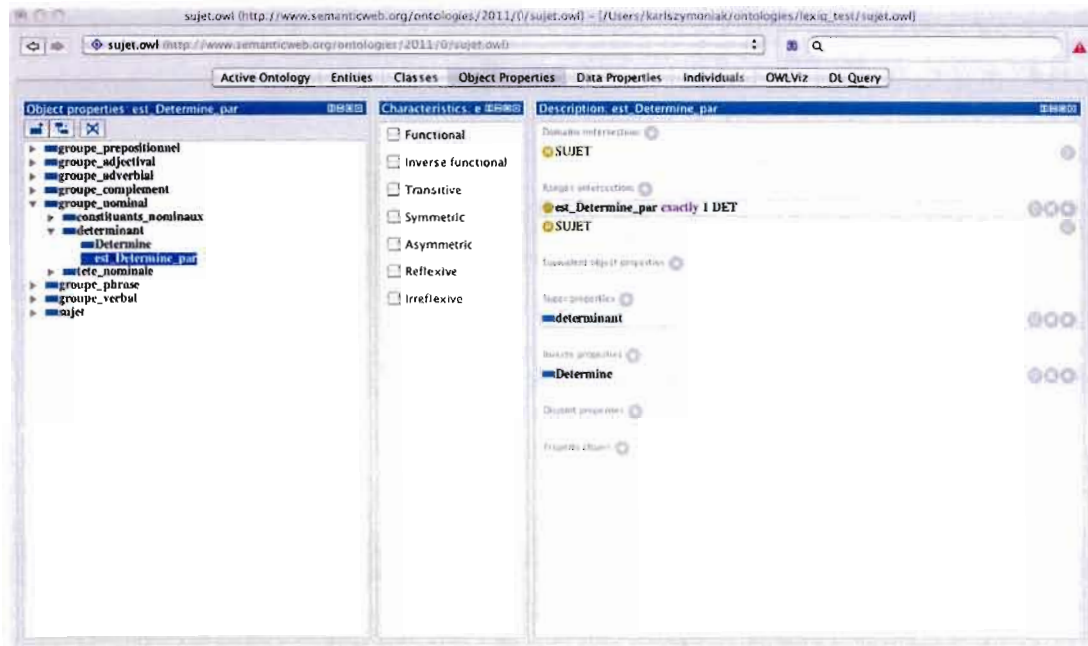


Figure 33 : Propriété liant le déterminant au nom.

Le logiciel Protégé permet de créer dès le départ une hiérarchie de classes. Toutefois, j'ai jugé plus pertinent de ne pas utiliser cette fonctionnalité du logiciel afin de pouvoir vérifier plus facilement la validité de mes propriétés et des inférences effectuées par le raisonneur de Protégé.

Un exemple d'utilisation de la fonction de hiérarchisation de classes est montré à la figure 34. La base de ma hiérarchie est montrée dans la partie droite de la figure 35. La

⁵⁴ Le raisonneur est un programme intégré à Protégé et qui permet à ce dernier d'effectuer les inférences entre classes, propriétés et individus.

hiérarchie inférée à partir de mes propriétés est montrée dans la partie gauche de la figure 35.

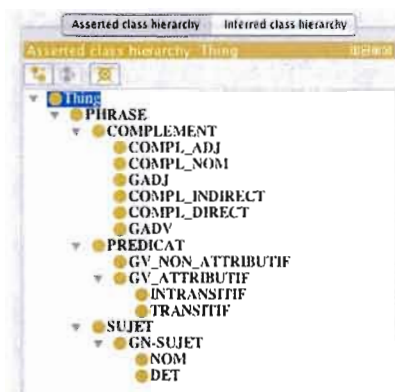


Figure 34 : Hiérarchisation des classes à l'aide du logiciel.

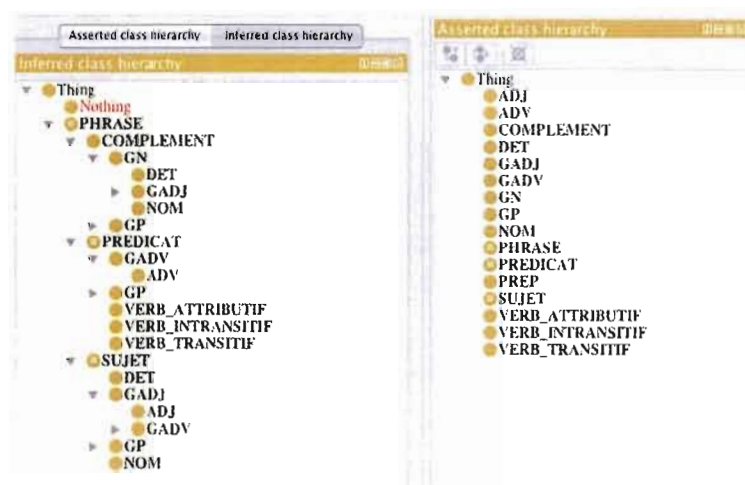


Figure 35 : Hiérarchie de mes classes après et avant l'utilisation du raisonneur.

On peut donc constater que les inférences effectuées par le raisonneur sur mon ontologie font que la phrase est constituée d'un sujet, d'un prédicat et de compléments. Mes propriétés bloquent le nombre de sujet et de prédicat à exactement un (dans le cas de la phrase de base), mais laisse le nombre de compléments libres. Cela signifie que la phrase

peut ne contenir aucun complément tout comme elle peut en contenir deux ou trois, voire plus.

2.5 Réalisation du programme de génération

Le générateur en tant que tel est constitué de deux programmes. Comme le montre la figure 36 qui est une représentation de mon algorithme.

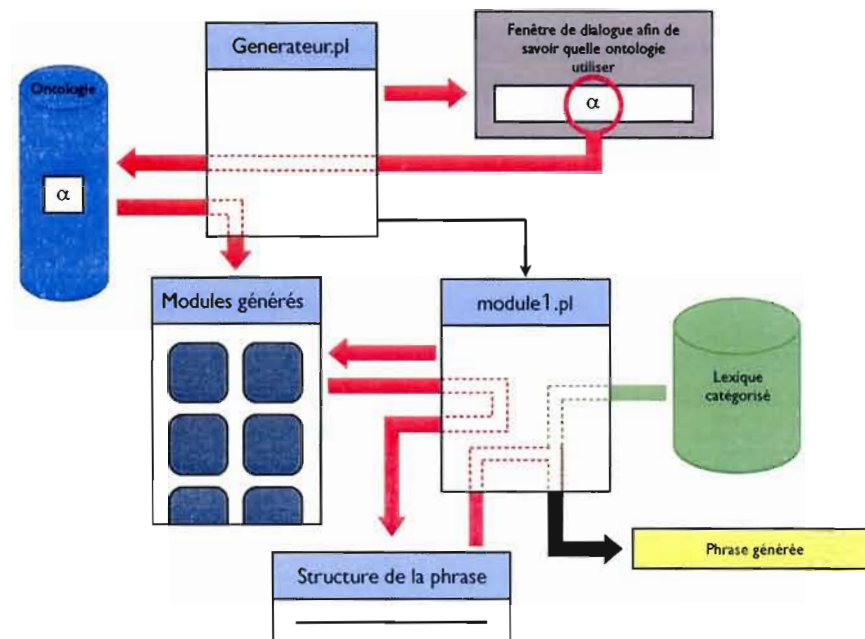


Figure 36 : Schéma de mon algorithme.

Le premier programme va aller chercher l'ontologie (α sur le schéma) à partir de laquelle je vais effectuer le travail de génération et la transformer en modules exploitables par le langage Perl. Chaque module correspond à une classe ou à une des propriétés établies dans l'ontologie. Seuls les individus ne sont pas représentés dans les modules. Cette récupération de l'ontologie s'effectue à l'aide de la boîte de dialogue apparaissant à figure 37.

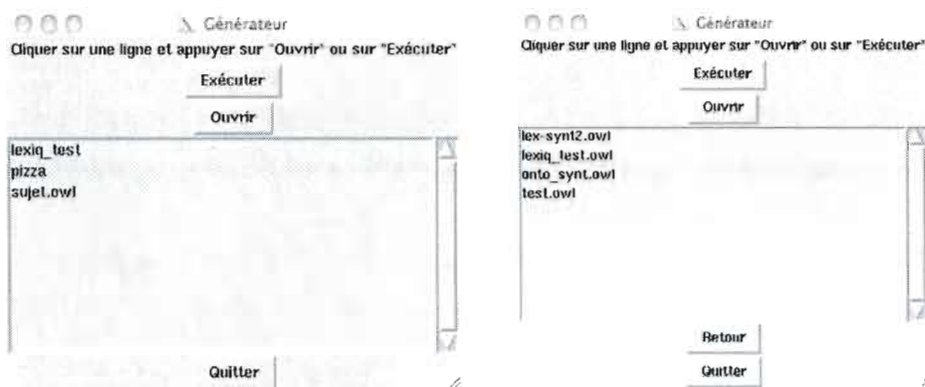


Figure 37 : Boîtes de dialogue du générateur.

La boîte de dialogue demande à l'utilisateur de choisir un répertoire (les éléments sans extension dans la liste) ou un fichier. Si l'utilisateur choisit un dossier, il lui faut ensuite cliquer sur 'Ouvrir' afin que le programme lui affiche le contenu du répertoire (partie droite de la figure). Dans ce cas, on peut constater l'apparition d'un bouton retour qui permet à l'utilisateur de remonter au répertoire précédent en cas de mauvaise manipulation. Quand l'utilisateur clique sur le bouton 'Exécuter', la récupération de l'ontologie et sa transformation en modules sont réalisées par le programme. Afin de réaliser cette transformation en modules, j'ai utilisé le CPAN nommé OWL2PERL.

Le second programme va, quant à lui, gérer la génération et l'utilisation des différents modules du premier programme. J'ai été obligé de créer un second script pour la génération et le contrôle des modules, car les modules sont chacun contenus dans un fichier externe qu'il faut importer dans le programme. Cette importation s'effectue dans l'entête du programme avant les instructions. Afin de savoir comment utiliser les modules, chacun d'eux contient, en fin de fichier, une explication sur son implémentation et son appel dans le programme, comme montré à la figure 38.

La figure 38 montre comment déclarer chaque classe dans le programme. Néanmoins, une adaptation des liens est nécessaire afin de recréer les liens de l'ontologie. Par exemple, à la ligne 191 de la figure 38, si l'on entre le code '`#someUR55`' directement, le programme est

⁵⁵ une URI (Uniform Resource Identifier, soit littéralement *identifiant uniforme de ressource*), doit permettre d'identifier une ressource de manière permanente, même si cette ressource est déplacée ou supprimée.

incapable de générer la hiérarchie et ne va générer que la classe VERB_INTRANSITIF. Ce qui peut poser quelques problèmes quant à la hiérarchie des classes. En remplaçant #someURI par la classe 'PREDICAT' on génère alors que le verbe intransitif est la tête du prédicat.

```

177 www::semanticweb::org::ontologies::2011::0::sujet::VERB_INTRANSITIF - an automatically generated owl class!
178
179 =cut
180
181 =head1 SYNOPSIS
182
183 use www::semanticweb::org::ontologies::2011::0::sujet::VERB_INTRANSITIF;
184 my $class = www::semanticweb::org::ontologies::2011::0::sujet::VERB_INTRANSITIF->new();
185
186 # get the uri for this class
187 my $uri = $class->uri;
188 # add object properties
189 # Make sure to use the appropriate OWL class!
190 # OWL::Data::OWL::Class used as an example
191 $class->add_gv_est_la_Tete(new OWL::Data::OWL::Class('#someURI'));
192
193 # get the gv_est_la_Tete object properties
194 my $gv_est_la_Tete_objects = $class->gv_est_la_Tete;
195
196 =cut
197
198 =head1 DESCRIPTION
199
200 <inherits from>:
201 L<OWL::Data::OWL::Class>
202
203
204

```

Figure 38 : Explication sur l'utilisation de la classe 'VERB_INTRANSITIF'.

C'est cette possibilité de hiérarchiser les classes qui permet de créer la structure de la phrase à générer. Un bout de cette structure est montré dans la partie résultat en 3.2 à la figure 46.

Une fois la structure de la phrase complète, le programme va récupérer mon lexique catégorisé lors de la recherche des patrons. Tout le vocabulaire ne va malheureusement pas être exploité comme expliqué dans la section 3.1.

Pour utiliser ce lexique, le programme va descendre dans la structure jusqu'aux noeuds terminaux et récupérer le vocabulaire correspondant aux terminaisons.

Lorsque l'ensemble des noeuds terminaux aura été récupéré, le programme les remplacera par le vocabulaire nécessaire récupéré dans mon lexique catégorisé. Le programme génère alors une phrase et l'affiche dans un fichier nommé 'structure.txt', dont plusieurs copies sont données dans le chapitre 3.2 Résultats et dans l'annexe 3.

Le lexique catégorisé ne sera pas récupéré dans un fichier global, mais dans divers sous-fichier, chaque sous-fichier correspondant à un noeud terminal de ma structure (nom, det, verbe transitif, etc.). Cette séparation est effectuée à la main, mais sera automatisée par la suite.

2.6 Conclusion

L'élaboration de ma méthodologie pour la réalisation de mon programme de génération m'aura causé beaucoup de tracas surtout avec les problèmes rencontrés au niveau de la génération des structures de traits par le LKB.

Néanmoins, même si mon programme et la réalisation de mon ontologie ont connu une série d'obstacles qui m'ont contraint à apporter des modifications dans les fondements même de mon projet, il n'en demeure pas moins que la réalisation de mes sous-programmes, de mon ontologie et du générateur, en lui-même, m'a appris énormément. Et en effet, même si l'obligation de restreindre mon générateur de phrases à un lexique limité est frustrant, les réalisations abouties, tout comme celles avortées, m'ont permis une compréhension plus fine du fonctionnement de mon ontologie et une envie de poursuivre son développement afin de générer également des phrases complexes, voire des textes.

CHAPITRE III

RÉSULTATS

Avant d'obtenir des résultats exploitables de la part de mon générateur de phrases, je suis passé au travers de diverses épreuves qui pour certaines n'ont pu être surmontées. C'est notamment le cas des problèmes liés à la génération de la structure de trait par le LKB, problème sur lequel je reviendrai plus en détail dans la première partie de ce chapitre avant d'exposer quelques phrases produites par mon générateur.

3.1 Problèmes rencontrés et solutions alternatives.

Depuis plusieurs mois, je travaille activement sur la réalisation de mon programme de génération, sur l'implémentation des outils nécessaires au traitement de mon corpus et à sa mise en forme, ainsi qu'à une meilleure maîtrise des divers logiciels utilisés.

J'ai commencé mon projet par la mise en forme du corpus et c'est à ce moment que les problèmes ont également démarré.

3.1.1 Les sous-programmes

Mon premier problème concernait la mise en forme du texte dans les fichiers de mon corpus. Comme le montre la figure 39⁵⁶, le fichier d'origine a une structure de mise en forme particulière.

En effet, hormis les lignes d'entête usuelles dans ce type de fichier, le reste du contenu est un peu moins conventionnel. Notamment l'intégration d'un grand nombre de lignes 'vides' dans le script. En réalité, ces lignes ne sont absolument pas vides, mais contiennent tout type de caractère invisible, allant de l'espace au retour à la ligne en passant à travers divers encodages. On y retrouve donc des structures lexicales telles que `\s` qui représente l'espace, le `\t` qui représente une tabulation, le `\n` qui fait passer à la ligne suivante ainsi que le `\r` qui renvoie au début de la ligne. Ce dernier symbole est particulièrement difficile à déceler quand il s'agit de le mettre en évidence dans un corpus, car il n'est pas des plus fréquents dans les documents. Les programmeurs évitent en général de le placer dans un document notamment si le document doit être analysé. Il m'aura fallu quelques tests avant de me rendre compte de sa présence dans le corpus.

```
<FILE-date="2008/01/01/19">
<article-nb="2008/01/01/19-1">
<filname-date="20080101"><AAMM="200801"><AAMMJ="20080101"><AAMMJJHH="2008010119">
<filname="SURF-0,2-3208,1-0,0-1"> o Un document confidentiel, que s&#38;#39;est procuré &#38;#34;Le Monde&#38;#34;,
trace les contours du désastre entraîné par le scrutin du 27 décembre, qui vient de plonger le Kenya dans une vague de
violences.
<filname="PROF-0,2-3208,1-0,0-1"> o
Un document confidentiel de sept pages, que s'est procuré Le Monde,
trace les contours du désastre entraîné par le scrutin du 27 décembre
2007, qui vient de plonger le Kenya dans une vague de violences. Le
texte recense les irrégularités constatées par les observateurs des
partis. Il a été annoté par un participant lors d'une réunion discrète
rassemblant des partis en lice. Cette réunion s'est tenue au cours de
la nuit du samedi 29 au dimanche 30 décembre, celle précédant l'annonce
des résultats de la présidentielle.

Au moins 15 personnes brûlées vives dans un
église dans l'ouest du Kenya

L'incendie criminel d'une église dans laquelle s'étaient réfugiés
plusieurs centaines de personnes cherchant à fuir les violences
post-électorales a fait entre quinze et trente morts à Eldoret, mardi
1er janvier, selon les sources. "Une foule énorme a attaqué l'église",
```

Figure 39 : En-tête et premières lignes d'un fichier texte du corpus.

⁵⁶ On notera qu'il s'agit d'une reproduction de la figure 23 de la page 48.

On peut également apercevoir sur la figure 40 un autre problème évoqué lors de la description du corpus.

Un problème récurrent et présent à chaque ligne qui suit la balise «<filename='PROF-»». Il s'agit de l'espace intégré juste avant la seconde lettre du premier mot de la ligne. Ce problème a été l'un des plus ennuyeux à traiter, car bien qu'il ne semble pas difficile à corriger, il pose de nombreux problèmes, principalement au niveau du A. En effet, la lettre-mot À m'a posé quelques difficultés en raison de la non-utilisation de l'accentuation sur les majuscules en France. Afin de corriger ce problème, j'ai contrôlé à la main l'ensemble des débuts de chaque phrase formée à l'aide du second sous-programme. Ce parcours m'a permis de récupérer les mots commençant par la lettre 'A' et contenant un espace. La liste de mots n'était pas très longue, je l'ai intégrée dans le programme afin de traiter ce problème.

Un second problème provenant des fichiers d'origine est le problème d'encodage. Comme on peut le constater à la quatrième ligne du fichier, un grand nombre de caractères «barbares⁵⁷» apparaissent. Ces caractères correspondent aux caractères accentués ou à des caractères spéciaux (apostrophes, etc.) qui sont écrits soit dans deux encodages différents soit dans une forme spécifique permettant leur interprétation par n'importe quel navigateur internet.

Dans le cas d'une écriture dans deux encodages différents, les caractères apparaissant à l'écran ne comporteraient pas de '&' ou de '#'. Cet affichage correspond donc à la seconde possibilité. Toutefois, la présence de ces caractères n'était pas limitée qu'aux entêtes d'articles, et j'ai donc dû les retransformer en caractère UTF-8.

Un autre problème survenu lors du traitement de ce corpus est l'inclusion du caractère de fin de phrase dans certaines abréviations. Par exemple le terme monsieur apparaissait sous la forme «M.». Cela a également provoqué quelques problèmes quand le point n'était pas après M, dans le cas de monsieur, mais après la première lettre d'un prénom tel que «E. Morin». Pour empêcher le retour à la ligne de Morin, j'ai effectué des tests sur quelques caractères avant le '.' afin de vérifier sa validité de fin de phrase. Notamment si le '.' est précédé d'une majuscule précédée d'un espace, alors il ne s'agit pas d'une fin de phrase. Si la

⁵⁷ Il ne s'agit absolument pas du nom de ces caractères, mais de la façon dont nous les appelions lors de mon année de Master à l'Université de Paris 3.

majuscule qui précède le point est également précédée d'un point, on peut supposer qu'il s'agit d'un acronyme et donc le point n'est pas une fin de phrase.

3.1.2 Le LKB

Le but de mon projet est de générer des phrases en utilisant une ontologie syntaxique ainsi qu'un lexique marqué. Ce dernier devait contenir les structures de traits des grammaires HPSG, ce qui aurait eu pour effet de lui intégrer des informations aussi bien syntaxiques que sémantiques.

Dans l'optique de réaliser ce lexique, j'ai décidé d'utiliser le logiciel LKB (pour plus de détail sur le logiciel, se référer à la section 1.3.3 du premier chapitre). Mon cours de linguistique informatique m'avait, en effet, permis de me familiariser avec ce logiciel. N'ayant pas eu besoin de la partie génération du logiciel lors de mes premiers contacts avec ce dernier, je ne savais pas encore que la génération connaissait quelques difficultés à s'effectuer. J'ai donc essayé de produire une grammaire qui me permettrait de générer un lexique contenant les structures de traits désirées.

La création d'une grammaire sémantique avec le LKB n'est pas chose aisée et après plusieurs essais plus ou moins réussis, je décidais de tenter une première génération afin d'avoir une idée plus précise des modifications à apporter à ma grammaire.

Ce fut là ma plus grande déception vis-à-vis de ce projet : la non-possibilité de générer une structure de traits pour mon lexique. En effet, le fait que le LKB ne génère plus les structures a été très déroutant dans la continuité de mon projet, et la recherche de solutions alternatives m'a obligé à revoir en profondeur l'intégralité de mon projet en me faisant perdre beaucoup de temps. J'ai passé presque deux mois à essayer de trouver une solution au problème de génération. Et malgré les informations fournies sur le site du logiciel, je n'ai pu réussir à faire fonctionner cette partie du logiciel.

J'ai finalement décidé d'abandonner l'emploi du LKB. Malheureusement l'abandon de ce programme m'a également obligé à abandonner l'utilisation de la HPSG, grammaire sur laquelle je bâtissais en grande partie mon projet.

Après plusieurs semaines de recherche, Roger Nkambou, mon codirecteur m'a parlé d'un logiciel qu'il avait utilisé avec une de ses étudiantes et qui pourrait remplacer le LKB. Sans donner une structure de trait tel que le LKB le faisait, ce logiciel permet de créer des relations entre les divers éléments de la phrase. J'aurais donc pu avoir un vocabulaire sémantiquement lié à adjoindre à mon ontologie. Les connaissances sémantiques de l'ontologie et les informations sémantiques de mon lexique m'auraient permis de pallier en partie l'abandon du LKB.

Après plusieurs recherches sur ce logiciel, il s'est avéré que ce logiciel ne traite que les textes de langue anglaise et ne traite pas le français. N'ayant pas de logiciel de remplacement, je pensais créer une seconde ontologie, lexicale cette fois-ci, qui viendrait compléter mon ontologie syntaxique. Mais cela allait m'entraîner encore plus de difficultés et me demander du temps supplémentaire que je n'avais pas.

À force de retourner le problème sous tous les angles, j'ai réalisé que l'idée d'une ontologie lexicale avait tout de même du sens, mais que je n'étais pas obligé pour autant d'en implémenter une nouvelle. En effet, j'étais déjà en train d'implémenter une ontologie. Je n'avais donc qu'à utiliser la structure de mon ontologie syntaxique en y intégrant un lexique au niveau des classes terminales. C'est de cette façon que j'intégrais dans mon ontologie syntaxique le vocabulaire étiqueté par le Tree-tagger comme des éléments terminaux de mes classes syntaxiques. Ainsi, la classe des noms contiendrait le lexique nominal et ainsi de suite pour chaque classe.

3.1.3 L'ontologie

L'implémentation de l'ontologie n'a pas été en soi d'une grande complexité, le tutorial de Matthew Horridge (2009) étant particulièrement bien écrit et d'une aide précieuse.

L'un des deux grands problèmes de l'implémentation d'une ontologie est le temps nécessaire à son implémentation et à la vérification de la validité des propriétés créées et des inférences engendrées.

Le second problème est l'instabilité du raisonneur utilisé par Protégé pour effectuer les inférences. En effet, lors de mes premiers essais, la mise en place de trop de propriétés pour un nombre limité de classes a entraîné un certain nombre de bugs dans le raisonneur qui m'obligeait à forcer la fermeture du logiciel. Cependant, je suis entièrement responsable de ces bugs, et un «allègement» des propriétés les a fait disparaître. Comme exemple d'allègement, je pense notamment au double lien entre les classes. Si une classe PREDICAT possède la propriété d'avoir un sujet de la classe SUJET, il n'est pas obligatoire de créer un lien disant que la classe SUJET a la propriété d'être le sujet de la classe PREDICAT. Surtout que ma propriété 'a_pour_Sujet' a pour propriété inverse 'est_le_Sujet_de'.

La simple déclaration est une règle dans la création de l'ontologie, le raisonneur créant lui même les inférences et l'explicitation des propriétés inverses.

3.1.4 Le programme

La programmation du générateur n'aura pas été très difficile, aux vues des problèmes antérieurs. Néanmoins, le premier problème rencontré a été celui de l'installation du module CPAN OWL2PERL. En effet, afin d'installer ce module, il faut au préalable installer d'autres modules dont je ne vous ferai pas la liste dans ce texte mais dont le nombre avoisine les quarante modules. Ce petit jeu d'installation m'aura pris plus d'une journée pour être finalisé.

De plus, le module OWL2PERL m'a également posé quelques problèmes. En effet, j'avais placé, comme expliqué plus haut, dans mon ontologie une partie de mon lexique afin de pouvoir générer directement depuis l'ontologie. L'exportation de l'ontologie vers les modules utilisable par Perl a fait disparaître mon lexique. En théorie, l'exportation était censée créer un module supplémentaire contenant toutes les instances de mon ontologie. Après maints essais, et recherche dans Protégé et dans les scripts d'exportation OWL2PERL

afin de contrôler que tous les packages avaient convenablement été installés, je décidais de ne pas pousser plus loin mes investigations et d'utiliser directement le fichier de lexique catégorisé.

C'est à ce moment que le programme a connu son plus gros bug. Alors qu'au début le programme me donnait la structure des syntagmes, celui-ci a tout à coup cessé de fonctionner et ne me sortait que des erreurs à l'affichage. Je n'ai malheureusement pas pensé à noter les erreurs obtenues ce qui aurait pu être utile si jamais je me retrouve dans la même situation.

Après plusieurs jours de recherche à ne pas comprendre d'où pouvaient provenir les erreurs obtenues, une discussion avec une de mes collègues sur un sujet tout autre m'a fait réaliser où j'aurais pu commettre une erreur lors de l'implémentation de mon programme.

Comme on peut le constater sur la figure 40, il y a une différence entre la seconde et la troisième ligne, en dehors du nom de la fonction appelée. Cette différence se tient au niveau de l'argument qui accompagne la fonction. Dans la seconde ligne, je précise que l'appel à la fonction 'a_pour_Sujet' demande un argument qui est de la class des sujets. À la troisième ligne, je dis simplement à mon programme que l'argument qui est lié à l'appel de la fonction contient une adresse de classe, sans préciser l'adresse de cette classe.

```
my $uri_predicat = $class_predicat -> uri;
$class_predicat -> add_a_pour_Sujet ($class_sujet);
$class_predicat -> add_est_Contenu_dans ('#someURI');
my $a_pour_Sujet_sujet = $class_predicat -> a_pour_Sujet;
my $est_Contenu_dans_phrase = $class_predicat -> est_Contenu_dans;
my $a_pour_Tete_verbe = $class_predicat -> gv_a_pour_Tete;
```

Figure 40 : URI.

Le programme ne pouvait donc pas aller chercher la classe correspondante à l'argument ce qui provoquait le bug général du programme.

Pensant l'erreur d'un tout autre genre, par exemple un besoin de remise à niveau de la version de Perl que j'utilise, j'ai effectué une mise à jour de Perl. Cette mise à niveau du logiciel n'a eu pour effet que de supprimer tous les liens d'importation des modules

(packages) extérieurs dans l'ensemble de mes programmes. En effet, lorsque l'on installe un complément à Perl, ce dernier crée des liens entre lui et le complément dans un dossier intitulé 'Lib'. La mise à jour de Perl efface l'ensemble des liens créés par les versions antérieures. Il a donc fallu que je réinstalle l'ensemble de mes modules, ce qui m'a pris pratiquement une journée.

3.2 Résultats

3.2.1 Les sous-programmes

L'ensemble des mes sous-programmes m'ont permis de réaliser une mise en forme de mon corpus telle que je la désirais autant au niveau des syntagmes de mes phrases qu'au niveau de mon lexique.

L'écriture d'un syntagme par ligne dans le fichier (figure 41) m'a permis de déterminer plus facilement les constituants afin de les implémenter dans l'ontologie.

```
<S>
<NP> PRO:DEM NOM </NP>
<VN> PRO:PER AUX:pres VER:ppe </VN>
<PP> PRP:det <PP> NOM <PP> PRP <NP> DET:ART NOM </NP> </PP> </PP> </PP>
<PP> PRP:det <NP> NOM NUM </NP> </PP>
<PP> PRP:det <NP> NOM NUM NOM </NP> </PP>
PUN
<NP> PRO:DEM </NP>
<VN> VER:ppe </VN>
<NP> DET:ART NOM </NP>
<PP> PRP:det <NP> NOM </NP> </PP>
<PP> PRP <NP> DET:ART NOM </NP> </PP>
</S>
```

Figure 41 : Syntagmes constituant une des phrases de mon corpus après traitement.

On peut tout de même constater que quelques erreurs dans l'étiquetage des syntagmes et du lexique sont encore présentes. Néanmoins, l'ontologie étant créée manuellement, j'ai corrigé les erreurs au fur et à mesure de leur apparition. Toutefois cette correction n'a été effectuée que sur l'ontologie et non pas dans le fichier de syntagmes. Par exemple à la troisième ligne, le second <PP> devrait être un <NP>.

Malgré les erreurs restantes, qui doivent tourner aux alentours des 5%, je considère que mes sous-programmes ont effectué un excellent travail en traitant un peu plus de cinquante et un mille deux cents articles. Ce qui représente un lexique d'environ quatorze millions de mots et sept millions cent mille syntagmes dans six cent trente milles phrases. Le temps de traitement de l'ensemble du corpus prend environ huit heures.

3.2.2 L'ontologie

Je suis très fier de mon ontologie qui bien qu'assez simpliste, par rapport à ce qu'elle devrait être, effectue les bonnes inférences et hiérarchise l'ensemble de mes classes telles que je le voulais.

Cette ontologie est la première que j'effectue et est également mon premier contact avec le logiciel Protégé. Bien que son utilisation ne soit pas des plus compliquée, le tutoriel fournit par Matthew Horridge (2009) me fut d'une très grande aide et m'a permis de mener à bien cette partie de mon projet dont un résultat est montré à la figure 42, il s'agit de la reprise de la figure 35 (p.67).

La hiérarchie de mes classes inférées se trouve à gauche et mes classes telles qu'introduites dans l'ontologie à droite de la figure. On peut constater que les classes réalisées sont relativement simples et qu'il me faudrait les développer afin de pouvoir générer des phrases complexes. Toutefois, la hiérarchie inférée montre une structure de phrase correcte.

Comme expliqué dans le chapitre de méthodologie, en 2.4.1, on peut voir, à l'aide des classes hiérarchisées, les différents constituants de la phrase. Par la suite, les propriétés vont limiter la phrase simple à un sujet et un prédicat, mais permettre plusieurs compléments.

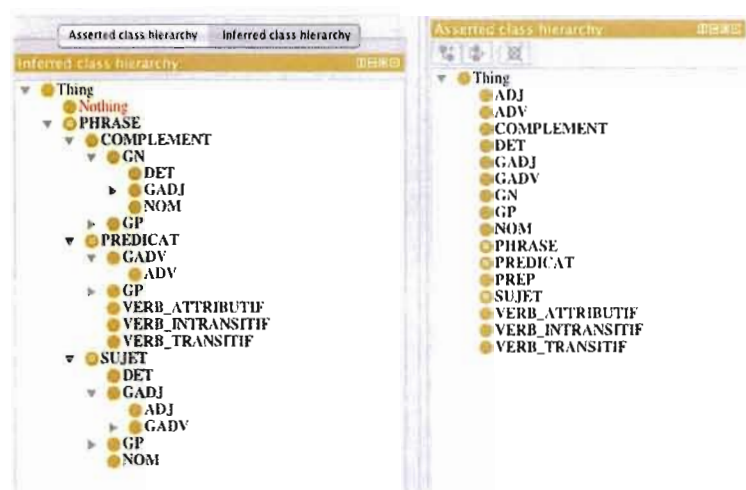


Figure 42 : Inférences réalisées sur les classes de mon ontologie.

Le logiciel Protégé permet également de générer un graphique de l'ontologie afin de voir les liens entre les diverses classes après le passage du raisonneur. Une vue schématique de mes classes telles qu'implémentées dans l'ontologie et une seconde après la réalisation des inférences sont montrées aux figure 43 et 44. La figure 44 explicite mon propos et montre les diverses relations entre les classes. Je tiens à m'excuser pour le fait de ne pas avoir pu remplacer les liens qui apparaissent sous le nom 'is a' alors qu'il devraient apparaître avec le nom de mes propriétés.

On peut constater qu'il manque quelques liens par exemple entre le GP et le GN sur ce schéma, le GP étant constitué d'une préposition et d'un GN. La propriété liant ces deux éléments existe dans l'ontologie et je ne comprends pas pourquoi ce lien et d'autres n'apparaissent pas sur le schéma. Ces liens sont toutefois pris en compte lors du passage aux modules. Si ce n'était pas le cas, mon verbe transitif n'aurait pas de complément lors de sa génération.

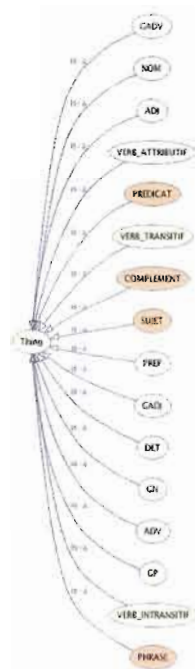


Figure 43 : Schéma de mes classes implémentées.

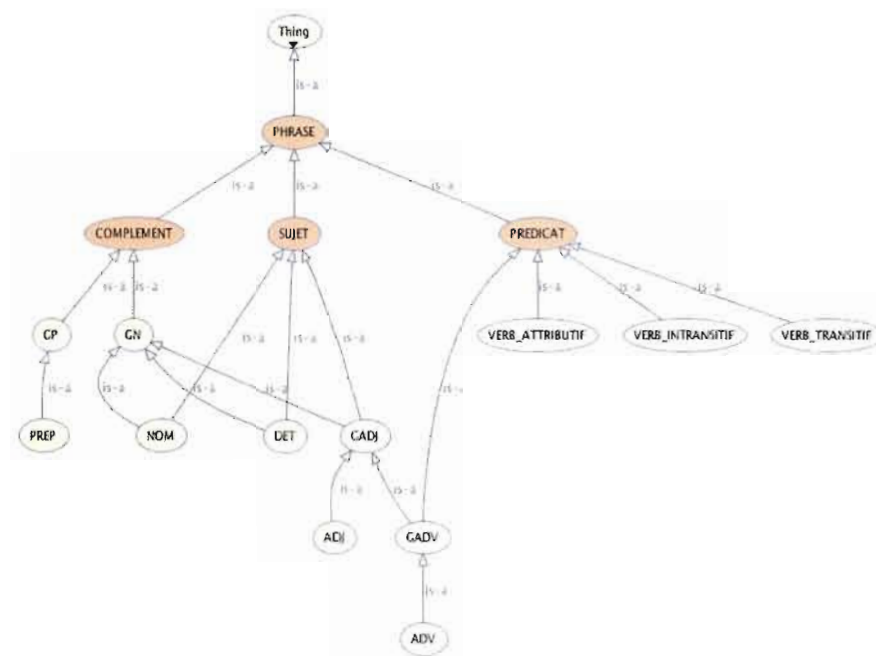


Figure 44 : Schéma de mes classes inférées.

3.2.3 Le générateur

Après avoir traité un corpus de quatorze millions de mots, construit une ontologie syntaxique et généré un lexique catégorisé, venait le temps de générer des phrases simples. Comme expliqué précédemment, le fait que mon vocabulaire ne soit pas pris en compte dans mon ontologie a posé quelques petits problèmes, mais mon programme était tout de même capable de générer la structure de la phrase, structure qu'il ne me restait plus qu'à compléter au niveau des noeuds terminaux. Un bout de cette structure est montré à la figure 45 et une structure complète se trouve dans l'annexe 3. Pour cette figure, j'ai fait disparaître quelques éléments générés, car ces éléments non nécessaires à la compréhension de la structure alourdisaient encore plus la figure et y tenaient une place non négligeable. Le fichier produit a une longueur qui triple avec l'intégration de ces éléments, comme on peut le constater à l'annexe 3 où la structure s'étale sur presque cinq pages.

```
-> www::semanticweb::org::ontologies::2011::0::sujet::PHRASE=HASH(0x1008050c8)
'Contient' => ARRAY(0x100b6f7d0)
  0 www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT=HASH(0x100b661b0)
    'a_pour_Sujet' => ARRAY(0x100b703e0)
      0 www::semanticweb::org::ontologies::2011::0::sujet::SUJET=HASH(0x100b66ed0)
        'Contient' => ARRAY(0x100966bb8)
          0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH(0x10082f128)
            'gn_Contient' => ARRAY(0x100967aa0)
              0 www::semanticweb::org::ontologies::2011::0::sujet::NOM=HASH(0x10082fdf8)
                'est_Determine_par' => ARRAY(0x1009690c0)
                  0 www::semanticweb::org::ontologies::2011::0::sujet::DET=HASH(0x100830ac8)
                    'Determine' => ARRAY(0x10096a650)
                      0 www::semanticweb::org::ontologies::2011::0::sujet::NOM=HASH(0x10082fdf8)
                        -> REUSED_ADDRESS
                      'gn_est_contenu_dans' => ARRAY(0x10096a110)
                        0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH(0x10082f128)
                          -> REUSED_ADDRESS
                      'strict' => 0
                      'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#DET'
                    'gn_est_contenu_dans' => ARRAY(0x1009686a0)
                      0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH(0x10082f128)
                        -> REUSED_ADDRESS
                      'gn_est_la_Tete_de' => ARRAY(0x100968be0)
                        0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH(0x10082f128)
                          -> REUSED_ADDRESS
                      'strict' => 0
                      'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#NOM'
                  1 www::semanticweb::org::ontologies::2011::0::sujet::DET=HASH(0x100830ac8)
```

Figure 45 : Partie d'une structure de phrase générée.

Afin de faciliter la compréhension de ce résultat, un peu écrasant au premier abord, je vais en expliciter la structure.

Le symbole ‘->’ sur la première ligne signifie que l’on va décrire la structure de la classe PHRASE. Le texte ‘www::semanticweb::org::ontologie::2010::0::sujet::PHRASE’ représente l’adresse exacte de la classe pour le programme, il s’agit de l’URI.

Pour le générateur, la structure de la phrase contient (fonction ‘Contient’) un tableau (‘=> ARRAY’, ligne 2) dont la première case (le 0 de la ligne 3) est de la classe des PREDICATS. Ce prédicat ‘a_pour Sujet’ le premier élément d’un tableau, élément de la classe ‘SUJET’. Cette classe ‘Contient’ un ‘GN’ qui lui même contient (‘gn_Contient’, ligne 8) un élément de la classe NOM.

Cet élément de la classe NOM⁵⁸ ‘est_Determine_par’ un DET (déterminant). Ce déterminant ‘Determine’ le NOM identifié précédemment et est contenu dans le GN qui contient le NOM (‘gn_est_contenu_dans’, ligne 20) et en est la tête (‘gn_est_la_Tete_de’, ligne 23).

Le 1 de la ligne 28 signifie que l’on est maintenant passé au deuxième élément qui compose le GN. Il s’agit d’un DET, le même que celui défini plus tôt.

On peut constater, même si le vocabulaire n’est pas présent, que la structure même de la phrase est «reconnaissable» et correspond à la structure des phrases simples telles que décrite dans le «*Vade-Mecum*» (Maisonneuve, 2003) et «*Le petit Grevisse*» (Grevisse, 2009).

Après avoir généré des structures acceptables, il me fallait ajouter à mes structures le lexique adapté qui constituerait mes phrases. Par récupération de mon lexique catégorisé que j’ai séparé manuellement en fonction de la catégorie du mot.

J’ai ensuite récupéré aléatoirement en fonction des noeuds terminaux dans les fichiers de vocabulaire, des éléments afin de former mes phrases. Les résultats sont montrés dans la figure 46. Un plus grand nombre de générations réalisées avec mon programme sont disponibles en deuxième partie de l’annexe 3.

⁵⁸ Pour la suite, je n’utilise plus que le nom de la classe en majuscule pour parler de cette dernière et non pas son adresse entière comme c’est le cas sur la figure.

Pour les résultats de mon générateur, j'aurais pu m'arrêter à la génération de la phrase, mais j'ai préféré récupérer la structure générée telle qu'elle apparaît dans la première partie de l'annexe 3 (où partiellement dans la figure 45) et l'intégrer afin de compléter ma génération. J'ai tout de même effectué un traitement sur cette structure afin qu'elle soit facilement lisible et compréhensible. Le résultat de ce traitement est montré à la figure 46 avec la phrase générée correspondante.

```

PHRASE
  PREDICAT
    VERB_TRANSITIF - revendique
    COMPLEMENT
      GN
        NOM - coup
        DET - le
  SUJET
    GN
      NOM - document
      DET - un

```

PHRASE : un document revendique le coup.

```

PHRASE
  PREDICAT
    VERB_INTRANSITIF - fuit
  SUJET
    GN
      NOM - service
      DET - un

```

PHRASE : un service fuit.

Figure 46 : Résultat de deux générations par mon programme.

On peut tout de même constater que le sujet est au même niveau que le verbe dans la structure alors qu'il devrait être au niveau du prédicat.

Cela s'explique facilement par les propriétés introduites dans mon ontologie. En effet, dans mon ontologie, j'ai indiqué que la tête de la phrase était un prédicat. Par la suite, j'ai indiqué que ce prédicat avait un sujet. Même si les inférences ont montré une bonne gestion de la hiérarchisation des éléments de ma phrase, le passage en modules pour l'intégration

dans mon programme a eu pour effet d'intégrer le sujet dans le prédicat. Une légère modification de l'ontologie devrait pouvoir arranger cela.

3.3 Conclusion

Malgré les difficultés techniques rencontrées, qui ont causé l'abandon d'une partie de mon projet, j'ai tout de même réussi à créer un générateur de phrases basé sur une ontologie syntaxique.

Un de mes regrets aura été l'impossibilité d'intégrer davantage d'aspect sémantique dans mon programme. Toutefois, je pense qu'une redéfinition de mes classes au sein même de mon ontologie ainsi qu'une révision/adaptation de mon lexique catégorisé aux futures classes de l'ontologie pourraient permettre la réalisation de phrases plus acceptables sémantiquement que celles créées à l'heure actuelle.

Mon programme remplit tout de même sa tâche, car juste avec l'ontologie, les phrases produites sont syntaxiquement correctes. Certes, il ne s'agit pour l'heure que de phrases simples, mais bien que ce mémoire touche à sa fin, le développement de mon générateur n'en est qu'à ses débuts.

CONCLUSION

Bien que la génération implique des connaissances linguistiques approfondies, le domaine de recherche dans lequel s'insère mon projet reste majoritairement aux mains des informaticiens. Il serait aisé de croire que, puisqu'il s'agit d'un programme informatique, son implémentation doit être réalisée par un programmeur. Mais le fait de produire un texte ou une phrase est un acte langagier et fait donc partie intégrante du domaine de la linguistique.

La création de ce générateur basé sur une ontologie syntaxique avait pour but de produire des phrases à la fois syntaxiquement et sémantiquement correctes. La partie sémantique devait être fournie par un lexique enrichi, à l'aide d'une structure de traits, généré par le LKB.

L'ensemble des problèmes rencontrés, notamment avec le LKB, a entraîné une perte de temps pour essayer de palier à la disparition de mon lexique typé. Ce temps perdu m'a obligé à concentrer mes efforts sur la réalisation de mon ontologie et donc sur la production de phrase syntaxiquement correctes et de ce fait abandonner la partie sémantique de ma génération. Même si les structures de phrases générées sont des structures de phrases simples, il n'en demeure pas moins qu'elles sont tout à fait correctes, et les problèmes relatifs au sens étaient, quant à eux, attendus. Pour y remédier, une catégorisation plus précise de mon vocabulaire et une redéfinition plus fine des classes syntagmatiques et des propriétés de

mon ontologie permettraient un choix plus approprié du lexique de mes phrases. Cela entraînerait une meilleure performance de la qualité sémantique des phrases générées.

Une implémentation de mes programmes de traitement de corpus effectuée par un informaticien m'aurait certainement fait gagner en efficacité et en temps. Toutefois, bien que mes programmes ne soient pas les plus performants du fait de mes connaissances limitées en informatique, ils répondent entièrement aux exigences de mon projet. Mais, d'une manière plus générale, cette absence de collaboration entre informaticiens et linguistes, nous oblige à implémenter à chaque fois des outils pour traiter et mettre en forme les données. Cette programmation des outils est souvent effectuée plusieurs fois par des équipes de recherche, qui ne sachant pas que l'outil existe, le reproduit. Ce phénomène ralentit le développement de la recherche en traitement automatique de la langue : une meilleure collaboration entre informaticiens et linguistes, ainsi qu'une plus grande diffusion des outils de traitement de données, ne pourrait qu'être bénéfique pour l'ensemble des deux parties.

Pour conclure, je pense qu'un développement plus approfondi de mon ontologie, dans le but de lui faire générer des phrases complexes, mais également pour renforcer les propriétés et les liens entre les classes qui la composent, devrait conduire mon générateur à une production plus sémantiquement. Pour cela je pense retravailler la taxinomie de mes classes ainsi que redéfinir mes classes elles-mêmes. Une intégration de métaclasse correspondant aux règles discursives pourrait permettre de passer de la production d'une phrase isolée à la production d'une succession de deux ou trois phrases liées sémantiquement par un thème commun et cohérente dans une même perspective discursive. Et cela dans le but de l'intégrer dans le générateur de dialogues du STI que je compte développer.

ANNEXES A

LES SOUS-PROGRAMMES

1 01-Projet_memoire_recup.pl

```
#!/usr/bin/Perl -w
use Tk;
use strict;

my $fichier;
my $out = "sorties";
my $ligne;

my %table;

# création de la fenêtre principale de dialogue
my $fenetre_principale_fichier = MainWindow -> new;
$fenetre_principale_fichier -> title ("Récupération du répertoire de départ:");
$fenetre_principale_fichier -> Label (
    -text => "Commencer par choisir le répertoire à récupérer."
) -> pack;

$fenetre_principale_fichier -> Label (
    -text => "Entrer le nom du fichier à ouvrir et appuyer sur la touche Enter",
    -font => "lucidasans-14",
) -> pack(
    -pady => 30);
```

```

my $texte = $fenetre_principale_fichier -> Entry (
    -background => "white",
    -font => "lucidasans-14",
    -foreground => "black",
    -textvariable => ¥$fichier,
    -relief => "groove",
    -takefocus => 1,
    -insertofftime => 175,
    -insertontime => 175
) -> pack(
    -expand => 1,
    -fill => "x",
    -padx => 20);

$texte -> bind('<Return>', ¥&recuperation);

$fenetre_principale_fichier -> Button (
    -text => "sortie du programme",
    -command => sub { exit }
) -> pack (
    -side => "bottom",
    -expand => 1,
    -fill => "both");

# on crée le bouton d'appel de la procédure
$fenetre_principale_fichier -> Button (
    -text => "Récupération du contenu du répertoire",
# on appelle la procédure de récupération
    -command => ¥&recuperation
) -> pack (
    -side => "bottom",
    -expand => 1,);

# on crée le bouton d'appel de la procédure de parcours
$fenetre_principale_fichier -> Button (
    -text => "Parcourir",
# on appelle la procédure de parcours
    -command => ¥&parcourir
) -> pack (
    -side => "bottom",
    -expand => 1,);

```

```
#fin de la fenetre principale de dialogue
MainLoop;
```

```
#procédure de parcours : cette procédure est utilisée dans le cas où l'utilisateur ne connaît
pas le nom du dossier.
```

```
sub parcourir {
```

```
# on fait disparaître la fenetre ouverte
$fenetre_principale_fichier -> withdraw();
```

```
# on récupère tous les noms des fichiers du répertoire dans un tableau
opendir (DIR,".");
my @files=readdir(DIR);
closedir (DIR);
```

```
my @ls;
my $i=0;
```

```
# on va récupérer le nom de tous les répertoires contenus dans le répertoire courant
foreach my $file (@files){
    if ((-d $file) && ($file!~/\.\.?/)) {
        $ls[$i]=$file;
        $i++; }}

```

```
# on crée une seconde fenetre de dialogue qui affichera l'ensemble des répertoires du dossier
courant
```

```
my $fenetre_secondaire_fichier = MainWindow -> new;
```

```
my $label = $fenetre_secondaire_fichier -> Label (
    -text => "Cliquer sur une ligne et appuyer sur ¥"Récupération des informations¥""
) -> pack;
```

```
$fenetre_secondaire_fichier -> Button (
    -text => "sortie du programme",
    -command => ¥&suite
) -> pack (
    -side => "bottom",
    -expand => 1,
    -fill => "both");
```

```

my $liste = $fenetre_secondaire_fichier -> Scrolled ("Listbox",
    -scrollbars => "e",
    -selectmode => 'single'
) -> pack(
    -side => "bottom",
    -expand => 1,
    -fill => "both");

$liste -> insert('end', @ls);

my $ligne_choisie = $fenetre_secondaire_fichier -> Button (
    -text => "Récupération des informations",
    -command => sub {
        # la ligne surlignée est la ligne choisie
        my @choix = $liste -> curselection();
        foreach (@choix) {
            # on récupère l'indice de la ligne d'annotation et on lance la procédure de récupération du
            # nom du dossier
            $liste->get($_);
            my $index=$_;
            &recuperation_info($ls[$index]);}}
    ) -> pack;

MainLoop;}

# procédure qui va récupérer le nom du répertoire et lancer la procédure de récupération des
# fichiers.
sub recuperation_info {
    $fichier = shift(@_);
    &recuperation;}

# procédure de nettoyage des lignes
sub nettoyage{
    $ligne = shift(@_);

    # on va remplacer les caractères n'apparaissant pas correctement
    $ligne =~ s/¥&#38¥;#39¥;/¥'/g;
    $ligne =~ s/¥&#38¥;#34¥;/¥"/g;
    $ligne =~ s/¥&#234¥;/¥é/g;
    $ligne =~ s/¥&#233¥;/¥è/g;
    $ligne =~ s/¥&lt;/¥</g;
    $ligne =~ s/¥&gt;/¥>/g;

```



```

$ligne =~ s/¥&eacute;/é/g;
$ligne =~ s/¥&Eacute;/É/g;
$ligne =~ s/¥&Egrave;/È/g;
$ligne =~ s/¥&egrave;/è/g;
$ligne =~ s/¥&agrave;/à/g;
$ligne =~ s/¥&Agrave;/À/g;
$ligne =~ s/¥&acirc;/â/g;
$ligne =~ s/¥&ugrave;/ù/g;
$ligne =~ s/¥&ucirc;/û/g;
$ligne =~ s/¥&uuml;/ü/g;
$ligne =~ s/¥&icirc;/î/g;
$ligne =~ s/¥&iuml;/ï/g;
$ligne =~ s/¥&ocirc;/ô/g;
$ligne =~ s/¥&ecirc;/ê/g;
$ligne =~ s/¥&ccedil;/ç/g;
$ligne =~ s/¥&euml;/ë/g;
$ligne =~ s/¥&#156;/oe/g;
$ligne =~ s/¥&#133;/.../g;
$ligne =~ s/¥&amp;/et/g;
$ligne =~ s/¥&#38;nbsp;/ /g;
$ligne =~ s/¥&et/g;
$ligne =~ s/¥r/ /g;

$ligne =~ s/¥n/_/g;

$ligne =~ s/^(¥s+)?("[B-Z]) ([[:lower:]]+) /$2$3 /;

$ligne =~ s/^(¥s*([[:upper:]]|É|È|À|Ô|Ê|Ù|Ç|I")([[:upper:]]|É|È|À|Ô|Ê|Ù|Ç|I' |,|")+$
$//;

$ligne =~ s/¥//g;
$ligne =~ s/<.*>//g;
$ligne =~ s/¥sM¥s?.¥s/ Monsieur /g;
$ligne =~ s/ Mme / Madame /g;

chomp($ligne);

$ligne =~ s/.*www¥..*¥....?//g;}

# procédure de lecture du répertoire
sub repertoire {
# on récupère l'argument d'appel de la procédure

```

```

my $rep=shift(@_);

my $fich="";
my $nb_ligne = 0;

# on récupère tous les noms des fichiers du répertoire dans un tableau
opendir (DIR,$rep);
my @files=readdir(DIR);
closedir (DIR);

# on va tester tous les noms des fichiers du tableau
foreach my $file (@files) {

# si on a . ou .., on passe au fichier suivant
next if $file =~ /^\.?$/;

# on met à jour le nom du fichier par rapport au répertoire courant.
$file=$rep."/".$file;

# on teste si le fichier est un répertoire si oui, on rappelle la procédure avec le nouveau
répertoire en argument
if (-d $file) {&repertoire($file);}

# s'il s'agit d'un fichier
if (-f $file) {
# si ce fichier se termine par txt
if ($file =~ /txt$/){
if ($file =~ /\. *-3224.* /){$fich = ".$out."/01-xml/Societe.xml";}
elsif ($file =~ /\. *-3208.* /){$fich = ".$out."/01-xml/AlaUne.xml";}
elsif ($file =~ /\. *-3232.* /){$fich = ".$out."/01-xml/Opinion.xml";}
elsif ($file =~ /\. *-3210.* /){$fich = ".$out."/01-xml/International.xml";}
elsif ($file =~ /\. *-829254.* /){$fich = ".$out."/01-xml/
ElectionAmericaine.xml";}
elsif ($file =~ /\. *-3214.* /){$fich = ".$out."/01-xml/Europe.xml";}
elsif ($file =~ /\. *-3234.* /){$fich = ".$out."/01-xml/Economie.xml";}
elsif ($file =~ /\. *-3236.* /){$fich = ".$out."/01-xml/Medias.xml";}
elsif ($file =~ /\. *-3244.* /){$fich = ".$out."/01-xml/Environnement.xml";}
elsif ($file =~ /\. *-3246.* /){$fich = ".$out."/01-xml/Culture.xml";}
elsif ($file =~ /\. *-651865.* /){$fich = ".$out."/01-xml/Technologie.xml";}
# elsif ($file =~ /\. *-3476.* /){$fich = ".$out."/01-xml/Cinema.xml";}
# elsif ($file =~ /\. *-3260.* /){$fich = ".$out."/01-xml/Livres.xml";}
# elsif ($file =~ /\. *-3404.* /){$fich = ".$out."/01-xml/Examens.xml";}
elsif ($file =~ /\. *-823353.* /){$fich = ".$out."/01-xml/Politique.xml";}

```

```

elseif ($file=~/*-987718.*/{ $fich="./.$out."/01-xml/Municipales-
cantonales.xml"}
else {}}

```

```

# on ouvre le fichier $file en lecture

```

```

    open (FILE, $file);

```

```

# on ouvre un fichier temporaire en rajout (toutes nouvelles informations seront inscrites à la
fin du fichier, on peut ainsi sauvegarder plusieurs fichiers en un seul)

```

```

    open (TMP, ">>$fich");

```

```

    my $surf = 0;

```

```

    my $prof = 0;

```

```

# tant qu'il y a des lignes dans le fichier, on va récupérer les lignes contenant les titres, celles
contenant les articles les nettoyer afin de résoudre les problèmes d'encodage et les copier
dans le fichier temporaire.

```

```

while ($ligne=<FILE>){
    if (($ligne=~/*<article-nb=/) && ($nb_ligne==1)) {
        print TMP "¥n¥t</Article>¥n";
        $surf = 0;
        $prof = 0;}
    if (($ligne=~/*<article-nb=/) && ($nb_ligne==0)) {
        $surf = 0;
        $prof = 0;
        $nb_ligne = 1;}
    if ($ligne=~/*<filename=¥"SURF/){
        print TMP "<Titre>¥n";
        $surf = 1;
        $prof = 0;}
    if ($ligne=~/*<filename=¥"PROF/){
        print TMP "¥n</Titre>¥n¥t<Article>¥n";
        $surf = 0;
        $prof = 1;}
    if ($surf == 1){
        &nettoyage($ligne);
        if ($ligne!~/^¥s*¥r?$/){print TMP "$ligne";}}
    if ($prof == 1){
        &nettoyage($ligne);
        if ($ligne!~/^¥s*¥r?$/){print TMP "$ligne";}}}

close TMP;
close FILE;}}

```

```
sub recuperation {  
  
    # on supprime le répertoire et tout ce qu'il contient s'il existe déjà  
    if (-e $out){system "rm -r $out";}  
  
    # on recrée le repertoire de sortie  
    if (!-d ".$out"){  
        mkdir("./.$out");  
        mkdir("./.$out."/01-xml");}  
  
    &repertoire($fichier);  
  
    #    system ("perl 02-Projet_memoire_phrases.pl");  
    #    system ("perl 03-Projet_memoire_mots.pl");  
  
    &suite;}  
  
sub suite {exit}
```

2 02-Projet_memoire_phrases.pl

```
#!/usr/bin/Perl -w
use Tk;
use strict;

my $ligne;

sub nettoyage {
    $ligne = shift(@_);

    $ligne =~ s/(¥.)_( _)+ +/$1 /g;
    $ligne =~ s/_( _)+ +/./ /g;
    $ligne =~ s/ +/ /g;
    $ligne =~ s/ M._? / Monsieur /g;
    $ligne =~ s/(¥s¥(?)(t|T)él._? / $1téléphone /g;
    $ligne =~ s/(¥s¥(?).éd._? / $1édition /g;
    $ligne =~ s/(¥s¥(?).p._? / $1page /g;
    $ligne =~ s/ Mme / Madame /g;
    $ligne =~ s/^ //;
    $ligne =~ s/¥^//g;
    $ligne =~ s/ ([:upper:]]¥. / monsieur /g;
    $ligne =~ s/^([[:upper:]]('| |[:upper:]]+)+ )([[:upper:]]([[:lower:]]+ )/$3/;
    $ligne =~ s/.*__+.*//;
    $ligne =~ s/^_ //;
    $ligne =~ s/_ / /g;
    $ligne =~ s/ +¥././g;
    $ligne =~ s/ *_ */g;}

mkdir("./sorties/02-phrases");
mkdir("./sorties/02-phrases/titres");
mkdir("./sorties/02-phrases/articles");

# on récupère tous les noms des fichiers du répertoire dans un tableau
opendir (DIR,"./sorties/01-xml");
my @files=readdir(DIR);
closedir (DIR);

my @ls;
my $i=0;
```

```

foreach my $file (@files){
    next if $file =~ /^%.%.$?$/;
    $ls[$i]=$file;
    $i++; }

my $fic;

while ($fic=<@ls>){
    $fic =~ s/%.xml//;
    my $phrases_titres = $fic."_phrases_titres";
    my $phrases_articles = $fic."_phrases_articles";
    $fic = "./sorties/01-xml/".$fic.".xml";

    open (FIC, $fic);
    open (TMP1, ">>./sorties/02-phrases/titres/$phrases_titres.tmp");
    open (TMP2, ">>./sorties/02-phrases/articles/$phrases_articles.tmp");

    my $titre = 0;
    my $article = 0;

    while ($ligne=<FIC>){

        if ($ligne=~<Titre>/){$titre = 1;}
        if ($ligne=~<%/Titre>/){$titre = 0;}
        if ($ligne=~<Article>/){$article = 1;}
        if ($ligne=~<%/Article>/){$article = 0;}

        if (($titre==1) && ($ligne!~/<Titre>/)){
            nettoyage($ligne);
            if ($ligne =~ %./){
                $ligne =~ s/(%.|%?|%)" ([:upper: ])/$1"%n$2/g;
                $ligne =~ s/(%.|%?|%)" ([:upper: ])/$1"%n$2/g;
                print TMP1 $ligne; }
            else {print TMP1 $ligne; }}

        if (($article==1) && ($ligne!~/<Article>/)){
            nettoyage($ligne);
            if ($ligne =~ %./){
                $ligne =~ s/(%.|%?|%)" ([:upper: ][:lower: ])/$1"%n$2/g;
                $ligne =~ s/(%.|%?|%)" ([:upper: ][:upper: ])/$1"%n$2/g;
                $ligne =~ s/(%.|%?|%)" ([:upper: ])/$1"%n$2/g;
                $ligne =~ s/(%.|%?|%)" ([:upper: ][:lower: ])/$1"%n$2/g;

```

```

        $ligne =~ s/(%.|%?|%) ([[[:upper:]]][:upper:]])/$1%$2/g;
        $ligne =~ s/(%.|%?|%) ([[[:upper:]]])/$1%$2/g;
        $ligne =~ s/(%.|%?|%) (%d)/$1%$2/g;
        $ligne =~ s/(%.|%?|%) ("[:upper:]][:lower:]])/$1%$2/g;
        $ligne =~ s/(%.|%?|%) ("[:upper:]][:upper:]])/$1%$2/g;
        $ligne =~ s/(%.|%?|%) ("[:upper:]])/$1%$2/g;
        $ligne =~ s/%. "%./"/g;
        $ligne =~ s/%. /%. %n/g;

        print TMP2 $ligne; }
    else{print TMP2 $ligne; }}

    close TMP2;
    close TMP1;
    close FIC;
}

my @phrase_tmp1;
my @phrase_tmp2;
my @tableau_phrases;

while ($fic=<@ls>){
    $fic =~ s/%.xml//;
    @phrase_tmp1 = ();
    @phrase_tmp2 = ();
    @tableau_phrases = ();
    my $phrases_titres = $fic."_phrases_titres";
    my $phrases_articles = $fic."_phrases_articles";

    open (TMP1,"./sorties/02-phrases/titres/$phrases_titres.tmp");
    open (PHRASE,">./sorties/02-phrases/titres/$phrases_titres.txt");

    my $i = 0;

    while ($ligne=<TMP1>){
        chomp($ligne);
        $phrase_tmp1[$i]=$ligne;
        $i++; }

    foreach $ligne (@phrase_tmp1) {print PHRASE "$ligne%$n";,;

    close PHRASE;

```

```

close TMP1;

system "rm ./sorties/02-phrases/titres/$phrases_titres.tmp";

open (TMP2,"./sorties/02-phrases/articles/$phrases_articles.tmp");
open (PHRASE,">./sorties/02-phrases/articles/$phrases_articles.txt");

$i = 0;

while ($ligne=<TMP2>){
    chomp($ligne);
    $phrase_tmp2[$i]=$ligne;
    $i++; }

foreach $ligne (@phrase_tmp2){
    if ($ligne =~ /\") {$ligne = "";}
    if ($ligne =~ /téléphone/) {$ligne = "";}
    if ($ligne =~ /édition/) {$ligne = "";}
    if ($ligne =~ /\^: ¥d¥d-¥d¥d/) {$ligne = "";}
    if ($ligne =~ /\^¥(.¥¥)¥.+$/ ) {$ligne = "";}
    if ($ligne =~ /¥(.+¥)/) {$ligne = "";}
    if ($ligne =~ /\^¥s*A¥s.*) {
        $ligne =~ s/A¥slors¥s/Alors /;
        $ligne =~ s/A¥svant /Avant/;
        $ligne =~ s/A¥sprès /Après/;
        $ligne =~ s/A¥sinsi /Ainsi/;
        $ligne =~ s/A¥sfin /Afin/;
        $ligne =~ s/A¥svec /Avec/;
        $ligne =~ s/A¥sux /Aux/;
        $ligne =~ s/A¥su /Au /;
    }
    elsif ($ligne =~ /\^(¥s*[[[:upper:]]]¥s((¥'|[[[:lower:]]].*)$)/) {
        $ligne = $1.$2;}
    $ligne =~ s/\^.* : ¥d{2},¥d{3} (¥d¥.)?.*//;
    $ligne =~ s/\^.*¥(.¥¥.gif¥).*$//;
    $ligne =~ s/\^.* : ¥d ¥d{3}.*//;
    $ligne =~ s/(([[[:upper:]]]|É|È|À|Ù|¥d|")+( '|, | )?) +$//;
    $ligne =~ s/\^¥d.*//;
    $ligne =~ s/\^ //g;
    $ligne =~ s/(¥!|¥?|¥.)¥.$/$1/;
    $ligne =~ s/¥.¥.$/¥.¥.¥./;
    if ($ligne!~/\^¥s*$/) {print PHRASE "$ligne¥n";}}

```



```
close PHRASE;
```

```
close TMP2;
```

```
system "rm ./sorties/02-phrases/articles/$phrases_articles.tmp";}
```

3 03-Projet_memoire_syntagm_tag

```
#!/usr/bin/Perl -w
use Tk;
use strict;

# on récupère tous les noms des fichiers du répertoire dans un tableau
opendir (DIR,"./sorties/02-phrases/articles");
my @files=readdir(DIR);
closedir (DIR);

my @ls;
my $i=0;

foreach my $file (@files){
    next if ($file =~ /^¥.¥.?$/);
    $ls[$i]=$file;
    $i++; }

my $fic;
my $sortie_tmp;
my $sortie_synt;
my $sortie_synt_tmp;
my $sortie_lem;
my $sortie_mot_tag;
my $sortie_phrase_lemme_tag;
my $sortie_phrase_cat_tag;
my @mot;

mkdir("./sorties/03-syntagmes");
mkdir("./sorties/03-syntagmes/lemmes");
mkdir("./sorties/03-syntagmes/cat");
mkdir("./sorties/03-syntagmes/tmp");
mkdir("./sorties/04-mots_tag");
mkdir("./sorties/05-phrases_tag");
mkdir("./sorties/05-phrases_tag/cat");
mkdir("./sorties/05-phrases_tag/lemmes");

while ($fic = <@ls>){

    my $fichier = "./sorties/02-phrases/articles/".$fic;
```

```

$sortie_tmp = "./sorties/03-syntagmes/tmp/".$fic;
$sortie_synt = "./sorties/03-syntagmes/cat/".$fic;
$sortie_synt_tmp = "./sorties/03-syntagmes/tmp/".$fic."_tmp";
$sortie_lem = "./sorties/03-syntagmes/lemmes/".$fic;
$sortie_mot_tag = "./sorties/04-mots_tag/".$fic;
$sortie_phrase_lemme_tag = "./sorties/05-phrases_tag/lemmes/".$fic;
$sortie_phrase_cat_tag = "./sorties/05-phrases_tag/cat/".$fic;

$sortie_tmp =~ s/_phrases_articles/_syntagmes/;
$sortie_synt =~ s/_phrases_articles/_syntagmes/;
$sortie_synt_tmp =~ s/_phrases_articles/_syntagmes/;
$sortie_lem =~ s/_phrases_articles/_syntagmes/;
$sortie_mot_tag =~ s/_phrases_articles/_mots_tag/;
$sortie_phrase_lemme_tag =~ s/_phrases_articles/_lemme_tag/;
$sortie_phrase_cat_tag =~ s/_phrases_articles/_cat_tag/;

print $fichier."¥n";

system("tree-tagger/cmd/tagger-chunker-french $fichier > $sortie_tmp");

my @tabfic;
$i = 0;

open (TMP, $sortie_tmp);
open (TMP2, ">$sortie_synt_tmp");

while (my $ligne = <TMP>){
    chomp ($ligne);
    $tabfic[$i] = $ligne;
    $i++;}

close TMP;

for my $j ( 0..$#tabfic ) {
    if ($tabfic[$j] =~ /(.)¥t(.+)¥t<unknown>/) {
        $tabfic[$j] =~ s/(.)(¥t.+¥t)<unknown>/$1$2$1/; }
    if (($tabfic[$j] =~ /<¥/NP>/) && ($tabfic[$j+1] =~ /<AP>/)) {
        my $garde = $tabfic[$j];
        my $k = $j;
        while ($tabfic[$k] !~ /<¥/AP>/) {
            $tabfic[$k] = $tabfic[$k+1];
            $k++;}
    }
}

```

```

    $tabfic[$k] = $garde;}
    if (($tabfic[$j] =~ /<¥/NP>/) && ($tabfic[$j+1] =~ /<¥/PP>/) && ($tabfic
[$j+2] =~ /<AP>/)) {
        my $garde = $tabfic[$j];
        my $garde2 = $tabfic[$j+1];
        my $k = $j;
        while ($tabfic[$k+2] !~ /<¥/AP>/) {
            $tabfic[$k] = $tabfic[$k+2];
            $k++;}
        $tabfic[$k] = "<¥/AP>";
        $tabfic[$k+1] = $garde;
        $tabfic[$k+2] = $garde2; }
    if (($tabfic[$j] =~ /<¥/NP>/) && ($tabfic[$j+1] =~ /<¥/PP>/) && ($tabfic
[$j+2] =~ /<¥/PP>/) && ($tabfic[$j+3] =~ /<AP>/)) {
        my $garde = $tabfic[$j];
        my $garde2 = $tabfic[$j+1];
        my $garde3 = $tabfic[$j+2];
        my $k = $j;
        while ($tabfic[$k+3] !~ /<¥/AP>/) {
            $tabfic[$k] = $tabfic[$k+3];
            $k++; }
        $tabfic[$k] = "<¥/AP>";
        $tabfic[$k+1] = $garde;
        $tabfic[$k+2] = $garde2;
        $tabfic[$k+3] = $garde3;}
    if ($tabfic[$j] =~ /<NP>/) {
        my $k = $j+1;
        my $ilyaunnom = 0;
        my $ilyaunadj = 0;
        while ($tabfic[$k] !~ /<¥/NP>/) {
            if ($tabfic[$k] =~ /NOM|NAM/) {$ilyaunnom = $k;}
            if (($tabfic[$k] =~ /ADJ/) && ($ilyaunadj == 0)) {$ilyaunadj
= $k;}

            $k++;}
        if (($ilyaunnom == 0) && ($ilyaunadj != 0)) {$tabfic[$ilyaunadj] =~
s/ADJ/NOM/;}}
    if (($tabfic[$j] =~ /ADV¥tne/) && ($tabfic[$j+1] =~ /entre/) && ($tabfic[$j
+2] =~ /ADV¥tpas/)) {$tabfic[$j+1] =~ s/PRP¥tentre/VER:pres¥tentrer/;}
    if (($tabfic[$j] =~ /<PP>/) && ($tabfic[$j+1] =~ /¥tPRP¥tentre/) &&
($tabfic[$j+2] =~ /<PP>/) && ($tabfic[$j+3] =~ /dans/) && ($tabfic[$j+4] =~ /<¥/PP>/))
{
        $tabfic[$j] = "<VN>";

```

```

$tabfic[$j+1] = ~ s/¥tPRP¥tentre/¥tVER:pres¥tentrer/ ;
$tabfic[$j+2] = "</VN>¥n".$tabfic[$j+2];
$tabfic[$j+4] = "";}

if (($tabfic[$j] = ~ /<¥/Ssub>/) && ($tabfic[$j+1] = ~ /<NP>/) && ($tabfic
[$j+2] = ~ /PRO:DEM/) && ($tabfic[$j+3] = ~ /<¥/NP>/) && ($tabfic[$j+4] = ~ /<NP>/) &&
($tabfic[$j+5] = ~ /PRO:REL/) && ($tabfic[$j+6] = ~ /<¥/NP>/) && ($tabfic[$j+7] = ~ /
<VN>/) && ($tabfic[$j+8] = ~ /PRO:PER/) && ($tabfic[$j+9] = ~ /VER:pres/) && ($tabfic[$j
+10] = ~ /ADV/)) {
    $tabfic[$j] = "";
    $tabfic[$j+3] = "";
    $tabfic[$j+4] = "";
    $tabfic[$j+6] = $tabfic[$j+8]."<¥/NP>";
    $tabfic[$j+8] = "";
    $tabfic[$j+9] = $tabfic[$j+9]. "¥n</VN>¥n</Ssub>¥n<VN>";}

if (($tabfic[$j] = ~ /<Ssub>/) && (($tabfic[$j+2] = ~ /<¥/Ssub>/) ||
($tabfic[$j+3] = ~ /<¥/Ssub>/))) {
    if ($tabfic[$j+2] = ~ /<¥/Ssub>/) {
        my $k = $j+3;
        while ($tabfic[$k] !~ /SENT|PUN/) {$k++;}
        $tabfic[$k] = $tabfic[$j+2]. "¥n". $tabfic[$k];
        $tabfic[$j+2] = "";}}

if ($tabfic[$j] = ~ /avoir|être/) {$tabfic[$j] = ~ s/VER/AUX/;}
if (($tabfic[$j] = ~ /pouvoir/) && ($tabfic[$j+3] = ~ /être/)) {
    $tabfic[$j] = ~ s/^(.)¥tVER.*$/$1/;
    $tabfic[$j] = $tabfic[$j]. "-être¥tADV¥tpouvoir-être";
    $tabfic[$j+1] = "";
    $tabfic[$j+2] = "";
    $tabfic[$j+3] = "";}

if (($tabfic[$j] = ~ /PUN/) && ($tabfic[$j+1] = ~ /(<¥/.+>)/)) {
    my $garde = $tabfic[$j+1];
    $tabfic[$j+1] = $tabfic[$j];
    $tabfic[$j] = $garde; }

if (($tabfic[$j] = ~ /AUX/) && ($tabfic[$j+1] = ~ /<¥/VN>/) && ($tabfic[$j
+2] = ~ /<(AP|VPpart)>/) && (($tabfic[$j+3] = ~ /VER/) || ($tabfic[$j+4] = ~ /VER/))) {
    my $garde = $tabfic[$j+1];
    my $k = $j+1;
    while ($tabfic[$k] !~ /<¥/(AP|VPpart)>/) {
        if ($tabfic[$k] = ~ /<.+>/) {$tabfic[$k] = "";}
        $k++; }
    $tabfic[$k] = $garde; }

```

```

    if (($tabfic[$j] =~ /<VN>/) && ($tabfic[$j+1] =~ /((i|I)ls?(E|e)lles?(n|N)
ous|(v|V)ous|(o|O)n|(j|J)(¥'|e))¥tPRO:PER/)) {
        my $garde = $tabfic[$j];
        $tabfic[$j] = "<NP>¥n".$tabfic[$j+1].¥n</NP>";
        $tabfic[$j+1] = $garde; }
    if (($tabfic[$j] =~ /parce/) && ($tabfic[$j+1] =~ /qu(e|¥')/)) {
        if ($tabfic[$j+1] =~ /que/) { $tabfic[$j] = "parce que¥tKON¥tparce
que";}

        else { $tabfic[$j] = "parce qu'¥tKON¥tparce que";}
        $tabfic[$j+1] = "";}
    if (($tabfic[$j] =~ /<¥/VN>/) && ($tabfic[$j+1] =~ /<AP|VN|NP|VPP.+>/)
&& ($tabfic[$j+2] =~ /VER/)) {
        my $motif = $tabfic[$j+1];
        $motif =~ s/</<¥//;
        my $k = $j+2;
        while ($tabfic[$k] !~ /$motif/) { $k++;}
        $tabfic[$k] = $tabfic[$j];
        $tabfic[$j+1] = "";
        $tabfic[$j] = "";}
    if ($tabfic[$j] =~ /<SENT>/) {
        if ($tabfic[$j+2] =~ /<¥/SENT>/) {
            if ($tabfic[$j+1] =~ /ADV/) {
                $tabfic[$j] = "<ADV>";
                $tabfic[$j+2] = "</ADV>";}
            elsif ($tabfic[$j+1] =~ /PUN/){
                $tabfic[$j] = "";
                $tabfic[$j+2] = "";}
            elsif ($tabfic[$j+1] =~ /KON/){
                $tabfic[$j] = "<Ssub>";
                $tabfic[$j+2] = "</Ssub>";}
            elsif ($tabfic[$j+1] =~ /PRO:(REL|DEM)/) {
                $tabfic[$j] = "<NP>";
                $tabfic[$j+2] = "</NP>";
                if ($tabfic[$j+3] =~ /<NP>/) {
                    $tabfic[$j+1] =~ s/PRO:REL/DET/;
                    $tabfic[$j+2] = "";
                    $tabfic[$j+3] = "";}}
            elsif ($tabfic[$j+1] =~ /PRP/){
                $tabfic[$j] = "<PP>";
                $tabfic[$j+2] = "</PP>";
                if ($tabfic[$j+3] =~ /<PP>/) {
                    $tabfic[$j+2] = "";

```

```

        $tabfic[$j+3] = "";}
    elseif ($tabfic[$j+3] =~ /<AP>/) {
        $tabfic[$j] = "<AP>";
        $tabfic[$j+2] = "";
        $tabfic[$j+3] = "";}}
elseif ($tabfic[$j+1] =~ /¥ten$/){
    $tabfic[$j] = "<PP>";
    $tabfic[$j+1] =~ s/PRO:PER/PRP/;
    if ($tabfic[$j+3] =~ /VN/) {
        $tabfic[$j+2] = "";
        my $k = $j+4;
        while ($tabfic[$k] !~ /<¥/VN>/) {$k++;}
        $tabfic[$k] = $tabfic[$k]. "¥n</PP>";}
    else {$tabfic[$j+2] = "</PP>";}}
elseif ($tabfic[$j+3] =~ /<¥/SENT>/) {
    if (($tabfic[$j+2] =~ /PUN/) && ($tabfic[$j+1] =~ /ADV/))
{
    $tabfic[$j] = "<ADV>";
    my $garde = $tabfic[$j+2];
    $tabfic[$j+2] = "</ADV>";
    $tabfic[$j+3] = $garde;}
    elseif (($tabfic[$j+2] =~ /PUN/) && ($tabfic[$j+1] =~ /
INT/)) {
        $tabfic[$j] = "<ADV>";
        $tabfic[$j+1] =~ s/INT/ADV/;
        my $garde = $tabfic[$j+2];
        $tabfic[$j+2] = "</ADV>";
        $tabfic[$j+3] = $garde;}
    elseif ($tabfic[$j+1] =~ /cela/) {
        $tabfic[$j] = "<NP>";
        $tabfic[$j+1] = $tabfic[$j+1]. "</NP>";
        $tabfic[$j+2] = "<VN>". $tabfic[$j+2];
        $tabfic[$j+2] =~ s/PRP/VER:pres/;
        if ($tabfic[$j+4] =~ /<(VP.+)>/) {
            my $motif = $1;
            my $k = $j+5;
            while ($tabfic[$k] !~ $motif) {$k++;}
            $tabfic[$k] = $tabfic[$k]. "¥n</VN>";
            $tabfic[$j+3] = "";}
        else {$tabfic[$j+3] = "</VN>";}}
    elseif (($tabfic[$j+1] =~ /PRP/) && ($tabfic[$j+4] =~ /<
(NP)>/)) {

```

```

        $tabfic[$j] = "<PP>";
        my $motif = "</".$1.">";
        my $k = $j+5;
        while ($tabfic[$k] !~ $motif) {$k++;}
        $tabfic[$k] = $tabfic[$k]."$n</PP>";
        $tabfic[$j+3] = "";}}
elseif ($tabfic[$j+4] =~ /<¥/SENT>/) {
    if (($tabfic[$j+1] =~ /¥ten$/ ) && ($tabfic[$j+2] =~ /¥tfaire
$/)) {

        $tabfic[$j+1] =~ s/(¥t).+(¥t)/$1PRP$2/;
        $tabfic[$j+2] =~ s/VER.+¥tfaire/NOM¥tfait/;
        if ($tabfic[$j+1] =~ /PRP/) {
            $tabfic[$j] = "<PP>";
            $tabfic[$j+4] = "</PP>";}}
elseif ($tabfic[$j+5] =~ /<¥/SENT>/) {
    if ($tabfic[$j+1] =~ /PRP/) {
        $tabfic[$j] = "<PP>";
        $tabfic[$j+5] = "</PP>";}}
elseif ($tabfic[$j+6] =~ /<¥/SENT>/) {
    if ($tabfic[$j+1] =~ /PRP/) {
        $tabfic[$j] = "<PP>";
        $tabfic[$j+6] = "</PP>";}}}
if (($tabfic[$j] =~ /exemple/) && ($tabfic[$j-1] =~ /¥tpar/)) {
    if ($tabfic[$j-1] =~ /Par/) {$tabfic[$j] =~ s/exemple¥tNOM
¥texemple/Par exemple¥tADV¥tpar exemple/;}
    else {$tabfic[$j] =~ s/exemple¥tNOM¥texemple/Par exemple¥tADV
¥tpar exemple/;}
    $tabfic[$j-1] = "";
    if ($tabfic[$j-2] =~ /<SENT>/) {
        $tabfic[$j-2] = "<ADV>";
        my $k = $j;
        while ($tabfic[$k] !~ /<¥/SENT>/) {$k++;}
        $tabfic[$k] = "</ADV>";}}

```

#j'ai été obligé d'éliminer cette partie du traitement, car elle faisait bugger le programme quand je traitais l'ensemble du corpus, mais pas pour des bouts séparés

```

#          if (($tabfic[$j] =~ /quelque/) && ($tabfic[$j+3] =~ /part/)) {
#              my $prem_lettre = "";
#              if ($tabfic[$j] =~ /(.)¥t.+¥tquelque/ ) {$prem_lettre = $1;}
#              .my $motif = "";
#              if ($tabfic[$j+2] =~ /<(.)>/) {$motif = "</".$1.">";}
#              my $k = $j+4;

```



```

# le problème semble venir de cette définition de $motif.
#           while ($tabfic[$k] !~ $motif) {$k++;}
#           $tabfic[$j-1] = "<ADV>";
#           $tabfic[$j] = $prem_lettre." part¥tADV¥t quelque part";
#           $tabfic[$j+1] = "</ADV>";
#           for my $l ( $j+2..$k ) {$tabfic[$l] = "";}

        if (($tabfic[$j] =~ /<¥/VPinf>/) && ($tabfic[$j+1] =~ /<VN>/)) {
            my $garde = $tabfic[$j];
            my $k = $j;
            while ($tabfic[$k] !~ /<¥/VN>/) {
                $tabfic[$k] = $tabfic[$k+1];
                $k++;}
            $tabfic[$k] = $garde;}
        if (($tabfic[$j] =~ /ADV/) && ($tabfic[$j+1] =~ /VER|AUX/) && ($tabfic[$j
+2] =~ /KON/)) {$tabfic[$j+2] =~ s/KON/ADV/;}
        if (($tabfic[$j] =~ /ADV/) && ($tabfic[$j+1] =~ /VER|AUX/) && ($tabfic[$j
+2] =~ /ADV/) && ($tabfic[$j+3] =~ /KON/)) {$tabfic[$j+3] =~ s/KON/ADV/;}
        print TMP2 $tabfic[$j]."\n"; }

#       while (my $cell = <@tabfic>) {print TMP2 $cell;}

close TMP2;

open (TMP2, $sortie_synt_tmp);
open (FIC, ">$sortie_synt");
open (PHTAG, ">$sortie_phrase_lemme_tag");
open (PHCAT, ">$sortie_phrase_cat_tag");
open (TAG, ">$sortie_mot_tag");
open (LEM, ">$sortie_lem");

my $motif = "";
my $nb_motif = 0;
my @fichier = <TMP2>;

for my $i ( 0..$#fichier ) {
    if ($fichier[$i] =~ /.+/){
        chomp($fichier[$i]);
        if (($fichier[$i] =~ /<(.*>/) && ($nb_motif == 0)) {
            $motif = "</"$1.">";
            $nb_motif = 1;}
        if (($fichier[$i] =~ /<s>/) || ($fichier[$i] =~ /<¥/s>/)) {

```

```

$motif = "";
$nb_motif = 0;
if ($fichier[$i] =~ /<s>/) {
    print FIC "<s>%n";
    print LEM "<s>%n";
    print PHCAT "<s> ";
}
else {
    print FIC "</s>%n";
    print LEM "</s>%n";
    print PHCAT "</s>%n";}}
if ($fichier[$i] =~ /PUN|SENT/) {
    @mot = split (/t/, $fichier[$i]);
    if ($mot[1] =~ /SENT/) {
        print TAG $fichier[$i]."%n";
        print PHTAG $mot[0]."|".$mot[1]."%n";
        print PHCAT $mot[0]."|".$mot[1]." ";
        print LEM $mot[0]."|".$mot[1]."%n";}
    else {
        print FIC $mot[1]."%n";
        print LEM $mot[0]."|".$mot[1]."%n";
        print TAG $fichier[$i]."%n";
        print PHTAG $mot[0]."|".$mot[1]." ";
        print PHCAT $mot[0]."|".$mot[1]." ";}}
if ($fichier[$i] !~ $motif) {
    if ($fichier[$i] !~ /</>/) {
        print TAG $fichier[$i]."%n";
        @mot = split (/t/, $fichier[$i]);
        if ($mot[1] =~ /SENT/) {}
        elsif ($mot[1] =~ /PUN/) {}
        else {
            print FIC $mot[1]." ";
            print LEM $mot[0]."|".$mot[1]." ";
            print PHTAG $mot[0]."|".$mot[1]." ";
            print PHCAT $mot[0]."|".$mot[1]." ";}}
    else {
        print FIC $fichier[$i]." ";
        print LEM $fichier[$i]." ";
        print PHCAT $fichier[$i]." ";}}
if (($fichier[$i] =~ $motif) && ($nb_motif == 1) && ($fichier[$i
+ 1] !~ $motif)) {
    print FIC $motif."%n";
    print LEM $motif."%n";

```

```
        $motif = "";
        $nb_motif = 0;}
    elif (($fichier[$i] =~ $motif) && ($nb_motif == 1) && ($fichier[$i
+1] =~ $motif)) {
        print FIC $motif." ";
        print LEM $motif." ";}}
    close TAG;
    close LEM;
    close PHTAG;
    close PHCAT;
    close FIC;
    close TMP2; }
system "rm -r ./sorties/03-syntagmes/tmp";
```

ANNEXE B

PROGRAMME DE GÉNÉRATION

1 Generateur.pl

```
#!/usr/bin/Perl -w
use Tk;
use strict;
use Text::Iconv;
use OWL2Perl;

my $converter = Text::Iconv->new("utf-8", "iso-8859-1");
my $converted = Text::Iconv->new("iso-8859-1", "utf-8");

chdir ("../../ontologies");

my $fenetre_seconde_fichier;

my %table;

my $fichier;
my $ligne;
my $chemin = ".";
my $liste;

my $nbfenetre = 0;

sub fenetre {
```

```

my @ls;
my $i=0;
my $dir = shift(@_);

if ($dir =~ /efface/) {
    my @rep = split (/\\/, $chemin);
    $chemin = "";
    for my $j (0..$#rep-1){$chemin = $chemin.$rep[$j]."/";}
    chdir(".");}
else {
    $dir = $dir."/";
    $chemin = $chemin.$dir;
    chdir($dir);}

opendir (DIR, ".");
my @files=readdir(DIR);
closedir (DIR);

foreach my $file (@files){
    if ((-e $file) && ($file!~/^%.%.?/)) {
        $ls[$i]=$file;
        $i++; }}

$fenetre_secondaire_fichier = MainWindow -> new;
$fenetre_secondaire_fichier -> title ($converter -> convert ("Générateur"));

my $label = $fenetre_secondaire_fichier -> Label (
    -text => $converter -> convert("Cliquer sur une ligne et appuyer sur
    ¥"Ouvrir¥" ou sur ¥"Exécuter¥")
    ) -> pack;

$fenetre_secondaire_fichier -> Button (
    -text => "Quitter",
    -command => ¥&suite
    ) -> pack (
        -side => "bottom",
        -expand => 1);

if ($nbfenetre > 0) {
    $fenetre_secondaire_fichier -> Button (
        -text => "Retour",

```

```

        -command => %&retour
    ) -> pack (
        -side => "bottom",
        -expand => 1);}

my $liste = $fenetre_secondaire_fichier -> Scrolled ("Listbox",
    -scrollbars => "e",
    -selectmode => 'single'
) -> pack(
    -side => "bottom",
    -expand => 1,
    -fill => "both");

$liste -> insert('end', @ls);

my $ligne_choisie = $fenetre_secondaire_fichier -> Button (
    -text => $converter -> convert("Exécuter"),
    -command => sub {
# la ligne surlignée est la ligne choisie
        my @choix = $liste -> curselection();
        foreach (@choix) {
# on récupère l'indice de la ligne d'annotation et on lance la procédure de
récupération du nom du dossier
            $liste->get($_);
            my $index=$_;
            &recuperation_info($ls[$index]);}}) -> pack;

my $ligne_ouverte = $fenetre_secondaire_fichier -> Button (
    -text => "Ouvrir",
    -command => sub {
# la ligne surlignée est la ligne choisie
        my @choix = $liste -> curselection();
        foreach (@choix) {
# on récupère l'indice de la ligne d'annotation et on lance la procédure de
récupération du nom du dossier
            $liste->get($_);
            my $index=$_;
            &ouverture($ls[$index]);}}) -> pack;

MainLoop;}

```

procédure qui va récupérer le nom du répertoire et lancer la procédure de récupération des fichiers.

```
sub ouverture {
    $fenetre_secondaire_fichier -> destroy;
    $fichier = shift(@_);
    $nbfenetre++;
    &fenetre($fichier);}

```

```
sub retour{
    $fenetre_secondaire_fichier -> destroy;
    &fenetre("efface");}

```

procédure qui va récupérer le nom du répertoire et lancer la procédure de récupération des fichiers.

```
sub recuperation_info {
    $fenetre_secondaire_fichier -> destroy;
    $fichier = shift(@_);
    $chemin =~ s/^..¥//;
    $chemin = $chemin.$fichier;
    #$chemin = "~/".$chemin;

```

#on récupère l'ontologie que l'on transforme en module pour Perl afin de faciliter son utilisation ultérieure.

```
system "owl2perl-generate-modules.pl -i //Users/karlszymoniak/ontologies/
$chemin";

```

```
my $chem = "//Users/karlszymoniak/ontologies/".$chemin;

```

```
#on revient dans le répertoire du projet où se trouve le second script.
chdir "../..Documents/PROJET_MEMOIRE/";

```

```
#on appelle le second script avec le chemin de l'ontologie en argument.
system "Perl module1.pl $chem";

```

```
&suite;
}

```

```
&fenetre($chemin);

```

```
sub suite {exit}

```

```
&fenetre;

```


2 module1.pl

```
#!/usr/bin/Perl -w
use Tk;
use strict;
use Text::Iconv;
use OWL::Data::OWL::Class;
use OWL::Utils;

#on récupère le chemin de l'ontologie.
my $chemin = shift;

# librairie à utiliser pour le programme :
use Lib '//Users/karlszymoniak/Perl-OWL2Perl/generated';
use www::semanticweb::org::ontologies::2011::0::sujet::PHRASE;
use www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT;
use www::semanticweb::org::ontologies::2011::0::sujet::SUJET;
use www::semanticweb::org::ontologies::2011::0::sujet::GN;
use www::semanticweb::org::ontologies::2011::0::sujet::NOM;
use www::semanticweb::org::ontologies::2011::0::sujet::DET;
use www::semanticweb::org::ontologies::2011::0::sujet::VERB_INTRANSITIF;
use www::semanticweb::org::ontologies::2011::0::sujet::VERB_TRANSITIF;
use www::semanticweb::org::ontologies::2011::0::sujet::COMPLEMENT;

# définition des classes liées aux librairies :
my $class_phrase = www::semanticweb::org::ontologies::2011::0::sujet::PHRASE -> new
();
my $class_predicat = www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT ->
new();
my $class_sujet = www::semanticweb::org::ontologies::2011::0::sujet::SUJET -> new();
my $class_gn = www::semanticweb::org::ontologies::2011::0::sujet::GN -> new();
my $class_nom = www::semanticweb::org::ontologies::2011::0::sujet::NOM -> new();
my $class_det = www::semanticweb::org::ontologies::2011::0::sujet::DET -> new();
my $class_gn_compl = www::semanticweb::org::ontologies::2011::0::sujet::GN -> new();
my $class_nom_compl = www::semanticweb::org::ontologies::2011::0::sujet::NOM -> new
();
my $class_det_compl = www::semanticweb::org::ontologies::2011::0::sujet::DET -> new
();
my $class_verb_intransitif;
my $class_verb_transitif;
```

```
my $class_compl = www::semanticweb::org::ontologies::2011::0::sujet::COMPLEMENT ->
new();
```

```
# définition des objets contenus dans chaque classe :
```

```
my $uri_phrase = $class_phrase -> uri;
$class_phrase -> add_Contient($class_predicat);
my $Contient_predicat = $class_phrase -> Contient;
```

```
my $uri_predicat = $class_predicat -> uri;
$class_predicat -> add_a_pour_Sujet ($class_sujet);
$class_predicat -> add_est_Contenu_dans ($class_phrase);
my $a_pour_Sujet_sujet = $class_predicat -> a_pour_Sujet;
my $est_Contenu_dans_phrase = $class_predicat -> est_Contenu_dans;
my $a_pour_Tete_verbe = $class_predicat -> gv_a_pour_Tete;
```

```
my $uri_sujet = $class_sujet -> uri;
$class_sujet -> add_est_le_Sujet_de($class_predicat);
$class_sujet -> add_Contient($class_gn);
my $est_le_Sujet_de_objects = $class_sujet -> est_le_Sujet_de;
my $Contient_objects = $class_sujet -> Contient;
```

```
my $uri_gn = $class_gn -> uri;
$class_gn -> add_gn_Contient($class_nom);
$class_gn -> add_gn_Contient($class_det);
my $gn_Contient_nom = $class_gn -> gn_Contient;
my $gn_Contient_det = $class_gn -> gn_Contient;
```

```
my $uri_gn_compl = $class_gn_compl -> uri;
$class_gn_compl -> add_gn_Contient($class_nom_compl);
$class_gn_compl -> add_gn_Contient($class_det_compl);
my $gn_Contient_nom_compl = $class_gn_compl -> gn_Contient;
my $gn_Contient_det_compl = $class_gn_compl -> gn_Contient;
```

```
my $uri_nom = $class_nom -> uri;
$class_nom -> add_gn_est_contenu_dans($class_gn);
$class_nom -> add_gn_est_la_Tete_de($class_gn);
$class_nom -> add_est_Determine_par($class_det);
my $gn_est_contenu_dans_objects = $class_nom -> gn_est_contenu_dans;
my $gn_est_la_Tete_de_objects = $class_nom -> gn_est_la_Tete_de;
my $est_Determine_par_objects = $class_nom -> est_Determine_par;
```

```
my $uri_nom_compl = $class_nom_compl -> uri;
```

```

$class_nom_compl -> add_gn_est_contenu_dans($class_gn_compl);
$class_nom_compl -> add_gn_est_la_Tete_de($class_gn_compl);
$class_nom_compl -> add_est_Determine_par($class_det_compl);
my $gn_est_contenu_dans_objects_compl = $class_nom_compl -> gn_est_contenu_dans;
my $gn_est_la_Tete_de_objects_compl = $class_nom_compl -> gn_est_la_Tete_de;
my $est_Determine_par_objects_compl = $class_nom_compl -> est_Determine_par;

my $uri_det = $class_det -> uri;
$class_det -> add_gn_est_contenu_dans($class_gn);
$class_det -> add_Determine($class_nom);
my $gn_est_contenu_dans_gn = $class_det -> gn_est_contenu_dans;
my $Determine_objects = $class_det -> Determine;

my $uri_det_compl = $class_det_compl -> uri;
$class_det_compl -> add_gn_est_contenu_dans($class_gn_compl);
$class_det_compl -> add_Determine($class_nom_compl);
my $gn_est_contenu_dans_gn_compl = $class_det_compl -> gn_est_contenu_dans;
my $Determine_objects_compl = $class_det_compl -> Determine;

my $uri_compl = $class_compl -> uri;
$class_compl -> add_compl_Contient($class_gn_compl);
my $compl_Contient = $class_compl -> compl_Contient;

#afin de choisir si le predicat contiendra un verbe transitif ou intransitif, on effectue un choix
aléatoire dont le résultat peut être 0 ou 1.
my $trans_ou_intrans = int(rand(2));

#si le choix est 0, on a un verbe intransitif, sinon il est transitif.
if ($trans_ou_intrans == 0) {
    $class_verb_intransitif = www::semanticweb::org::ontologies::
2011::0::sujet::VERB_INTRANSITIF -> new();
    my $uri_verb_intransitif = $class_verb_intransitif -> uri;
    $class_verb_intransitif -> add_gv_est_la_Tete($class_predicat);
    my $est_la_Tete_verb_intransitif = $class_verb_intransitif -> gv_est_la_Tete;
    $class_predicat -> add_gv_a_pour_Tete ($class_verb_intransitif);}
else {
    $class_verb_transitif = www::semanticweb::org::ontologies::
2011::0::sujet::VERB_TRANSITIF -> new();
    my $uri_verb_transitif = $class_verb_transitif -> uri;
    $class_verb_transitif -> add_gv_est_la_Tete($class_predicat);
    $class_verb_transitif -> add_est_Complemente_par($class_compl);
    my $est_la_Tete_verb_transitif = $class_verb_transitif -> gv_est_la_Tete;

```

```
my $est_Complemente_par = $class_verb_transitif -> est_Complemente_par;
$class_predicat -> add_gv_a_pour_Tete ($class_verb_transitif);}
```

#on récupère la structure de la phrase dans le fichier 'structure'

```
open (FICH, ">structure");
print FICH $class_phrase;
close FICH;
```

#récupération du vocabulaire catégorisé par Tree-Tagger.

```
open (TEST, "test_det.txt");
my @det = <TEST>;
close TEST;
open (TEST, "test_nom.txt");
my @nom = <TEST>;
close TEST;
open (TEST, "test_verbt.txt");
my @verbt = <TEST>;
close TEST;
open (TEST, "test_verbi.txt");
my @verbi = <TEST>;
close TEST;
```

#on va aller chercher l'ensemble des noeuds terminaux afin de leur donner une valeur (un lexique) à partir du vocabulaire catégorisé.

```
open (FICH, "structure");
open (STRUC, ">abc");
```

```
my $lign;
```

```
while (my $ligne = <FICH>) {
    if ($ligne =~ /(¥s*).*#(.*)/) {
        my $terminaison = $1.$2;
        $terminaison =~ s/'//;
        print STRUC $terminaison."¥n"}
}
```

```
close STRUC;
close FICH;
```

```
system "rm structure";
```

```
open (STRUC, "abc");
```

```

my @struc=<STRUC>;
close STRUC;

system "rm abc";

my $index_det;
my $index_nom;
my $index_verbt;
my $index_verbi;

for my $i ( 0 .. $#struc ){
    if ($struc[$i] =~ /DET/) {
        chomp ($struc[$i]);
        $index_det = int(rand($#det+1));
        $struc[$i] = $struc[$i]." - ".$det[$index_det];}
    elsif ($struc[$i] =~ /NOM/) {
        chomp ($struc[$i]);
        $index_nom = int(rand($#nom+1));
        $struc[$i] = $struc[$i]." - ".$nom[$index_nom];}
    elsif ($struc[$i] =~ /VERB_TR/) {
        chomp ($struc[$i]);
        $index_verbt = int(rand($#verbt+1));
        $struc[$i] = $struc[$i]." - ".$verbt[$index_verbt];}
    elsif ($struc[$i] =~ /VERB_IN/) {
        chomp ($struc[$i]);
        $index_verbi = int(rand($#verbi+1));
        $struc[$i] = $struc[$i]." - ".$verbi[$index_verbi];}
    else {} }

for my $i ( 0 .. $#struc ){
    if ($struc[$i] =~ /. * - .*¥t.*¥t.*/) { $struc[$i] =~ s/^(.* - .* )¥t.*¥t.*$/$1/; }
    if ($struc[$i] !~ /¥n/) { $struc[$i] = $struc[$i]."¥n"; } }

open (FICH, ">structure.txt");
for my $i ( 0 .. $#struc ) { print FICH $struc[$#struc-$i]; }

if ($struc[4] !~ /VER/) {
    my $gard = $struc[8];
    my $j = 8;
    while ($j>=5) {
        $struc[$j] = $struc[$j-1];
        $j--;}
}

```

```
$struc[4] = $gard;}

chomp($struc[$#struc]);
print FICH "¥n¥n".$struc[$#struc].": ";
for my $i ( 0 .. $#struc ) {
    if ($struc[$i] =~ / - /) {
        my @mot = split (/ - /, $struc[$i]);
        chomp($mot[1]);
        print FICH " ".$mot[1];}}
print FICH ".";

close FICH;
```

ANNEXE C

EXEMPLE DE RÉSULTATS

1 Génération de la structure de base.

```
-> www::semanticweb::org::ontologies::2011::0::sujet::PHRASE=HASH(0x1008050c8)
  'Contient' => ARRAY(0x100b6c898)
    0 www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT=HASH
      (0x100b3ca48)
        'a_pour_Sujet' => ARRAY(0x100b704a8)
          0 www::semanticweb::org::ontologies::2011::0::sujet::SUJET=HASH
            (0x100b639a0)
              'Contient' => ARRAY(0x100b731d8)
                0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
                  (0x100b655f8)
                    'gn_Contient' => ARRAY(0x100b740e8)
                      0 www::semanticweb::org::ontologies::2011::0::sujet::NOM=HASH
                        (0x100b662c8)
                          'est_Determine_par' => ARRAY(0x100966390)
                            0 www::semanticweb::org::ontologies::2011::0::sujet::DET=HASH
                              (0x100b67f98)
                                'Determine' => ARRAY(0x1009679f8)
                                  0 www::semanticweb::org::ontologies::
2011::0::sujet::NOM=HASH
                                                                    (0x100b662c8)

-> REUSED_ADDRESS
'gn_est_contenu_dans' => ARRAY(0x1009674b8)
  0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
```

```

(0x100b655f8)

-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b684d8)
'_options' => HASH(0x100b66868)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b683d0)
'_data' => HASH(0x100b683b8)
    empty hash
'_objects' => HASH(0x100b68448)
    empty hash
'_predicates' => HASH(0x100b68478)
    empty hash
'_subjects' => HASH(0x100b68400)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
sujet.owl#DET'
'gn_est_contenu_dans' => ARRAY(0x100b74d48)
    0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
        (0x100b655f8)

-> REUSED_ADDRESS
'gn_est_la_Tete_de' => ARRAY(0x100965eb0)
    0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
        (0x100b655f8)

-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b66808)
'_options' => HASH(0x100b65b98)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b66700)
'_data' => HASH(0x100b666e8)
    empty hash
'_objects' => HASH(0x100b66778)
    empty hash
'_predicates' => HASH(0x100b667a8)
    empty hash
'_subjects' => HASH(0x100b66730)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
sujet.owl#NOM'
1 www::semanticweb::org::ontologies::2011::0::sujet::DET=HASH
    (0x100b67f98)

-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b65b38)

```



```

'_options' => HASH(0x100b64ef0)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b65a30)
'_data' => HASH(0x100b65a18)
    empty hash
'_objects' => HASH(0x100b65aa8)
    empty hash
'_predicates' => HASH(0x100b65ad8)
    empty hash
'_subjects' => HASH(0x100b65a60)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#GN'
'est_le_Sujet_de' => ARRAY(0x100b72e90)
0 www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT=HASH
    (0x100b3ca48)
-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b64e90)
'_options' => HASH(0x100b636e8)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b63d60)
'_data' => HASH(0x100b63d48)
    empty hash
'_objects' => HASH(0x100b63dd8)
    empty hash
'_predicates' => HASH(0x100b64e30)
    empty hash
'_subjects' => HASH(0x100b63d90)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#SUJET'
'est_Contenu_dans' => ARRAY(0x100b709e8)
0 www::semanticweb::org::ontologies::2011::0::sujet::PHRASE=HASH
(0x1008050c8)
-> REUSED_ADDRESS
'gv_a_pour_Tete' => ARRAY(0x10096ae78)
0 www::semanticweb::org::ontologies::2011::0::sujet::VERB_TRANSITIF=HASH
    (0x100968d00)
'est_Complemente_par' => ARRAY(0x10096a740)
0 www::semanticweb::org::ontologies::2011::0::sujet::COMPLEMENT=HASH
    (0x100b6a0d0)
'compi_Contient' => ARRAY(0x100968970)
0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
    (0x100b68c58)

```

```

'gn_Contient' => ARRAY(0x100b74508)
0 www::semanticweb::org::ontologies::2011::0::sujet::NOM=HASH
(0x100b69e30)
'est_Determine_par' => ARRAY(0x100966e88)
0 www::semanticweb::org::ontologies::
2011::0::sujet::DET=HASH
(0x100b69fc8)
'Determine' => ARRAY(0x100968148)
0 www::semanticweb::org::ontologies::
2011::0::sujet::NOM=HASH
(0x100b69e30)
-> REUSED_ADDRESS
'gn_est_contenu_dans' => ARRAY(0x100968100)
0 www::semanticweb::org::ontologies::
2011::0::sujet::GN=HASH
(0x100b68c58)
-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b6a0b8)
'_options' => HASH(0x100b69f80)
'Storage' => RDF::Core::Storage::Memory=HASH
(0x100b69f98)
'_data' => HASH(0x100b69ef0)
empty hash
'_objects' => HASH(0x100b69ff8)
empty hash
'_predicates' => HASH(0x100b6a058)
empty hash
'_subjects' => HASH(0x100b69f38)
empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
sujet.owl#DET'
'gn_est_contenu_dans' => ARRAY(0x100966df8)
0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
(0x100b68c58)
-> REUSED_ADDRESS
'gn_est_la_Tete_de' => ARRAY(0x100966e40)
0 www::semanticweb::org::ontologies::2011::0::sujet::GN=HASH
(0x100b68c58)
-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b69f20)
'_options' => HASH(0x100b68dc0)

```

```

'Storage' => RDF::Core::Storage::Memory=HASH(0x100b68dd8)
'_data' => HASH(0x100b68d30)
    empty hash
'_objects' => HASH(0x100b69e60)
    empty hash
'_predicates' => HASH(0x100b69ec0)
    empty hash
'_subjects' => HASH(0x100b68d78)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
sujet.owl#NOM'
1 www::semanticweb::org::ontologies::2011::0::sujet::DET=HASH
    (0x100b69fc8)
-> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100b68d60)
'_options' => HASH(0x100b68538)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b68c70)
'_data' => HASH(0x100b68bb0)
    empty hash
'_objects' => HASH(0x100b68cb8)
    empty hash
'_predicates' => HASH(0x100b68d00)
    empty hash
'_subjects' => HASH(0x100b68c40)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#GN'
'model' => RDF::Core::Model=HASH(0x100b6a670)
'_options' => HASH(0x100b6a118)
'Storage' => RDF::Core::Storage::Memory=HASH(0x100b6a568)
'_data' => HASH(0x100b6a550)
    empty hash
'_objects' => HASH(0x100b6a5e0)
    empty hash
'_predicates' => HASH(0x100b6a610)
    empty hash
'_subjects' => HASH(0x100b6a598)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
sujet.owl#COMPLEMENT'

```

```

'gv_est_la_Tete' => ARRAY(0x10096a200)
  0 www::semanticweb::org::ontologies::2011::0::sujet::PREDICAT=HASH
                                     (0x100b3ca48)
    -> REUSED_ADDRESS
'model' => RDF::Core::Model=HASH(0x100969210)
'_options' => HASH(0x100b6a6d0)
'_Storage' => RDF::Core::Storage::Memory=HASH(0x100969108)
'_data' => HASH(0x1009690f0)
    empty hash
'_objects' => HASH(0x100969180)
    empty hash
'_predicates' => HASH(0x1009691b0)
    empty hash
'_subjects' => HASH(0x100969138)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/
                                     sujet.owl#VERB_TRANSITIF'
'model' => RDF::Core::Model=HASH(0x100b63688)
'_options' => HASH(0x100b63220)
'_Storage' => RDF::Core::Storage::Memory=HASH(0x100b635e0)
'_data' => HASH(0x100b63190)
    empty hash
'_objects' => HASH(0x100b63538)
    empty hash
'_predicates' => HASH(0x100b63610)
    empty hash
'_subjects' => HASH(0x100b635f8)
    empty hash
'strict' => 0
'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#PREDICAT'
'model' => RDF::Core::Model=HASH(0x100b58e98)
'_options' => HASH(0x100a6f890)
'_Storage' => RDF::Core::Storage::Memory=HASH(0x100b58d18)
'_data' => HASH(0x100b58d60)
    empty hash
'_objects' => HASH(0x100b58dc0)
    empty hash
'_predicates' => HASH(0x100b58df0)
    empty hash
'_subjects' => HASH(0x100b58d78)
    empty hash

```

'strict' => 0

'type' => 'http://www.semanticweb.org/ontologies/2011/0/sujet.owl#PHRASE'

2 Génération de phrase.

PHRASE
 PREDICAT
 VERB_TRANSITIF - revendique
 COMPLEMENT
 GN
 NOM - coup
 DET - le
 SUJET
 GN
 NOM - document
 DET - un

PHRASE : un document revendique le coup.

PHRASE
 PREDICAT
 VERB_INTRANSITIF - fuit
 SUJET
 GN
 NOM - service
 DET - un

PHRASE : un service fuit.

PHRASE
 PREDICAT
 VERB_INTRANSITIF - cherche
 SUJET
 GN
 NOM - ministre
 DET - le

PHRASE : le ministre cherche.

PHRASE
PREDICAT
VERB_INTRANSITIF - fuit
SUJET
GN
NOM - coup
DET - un

PHRASE : un coup fuit.

PHRASE
PREDICAT
VERB_INTRANSITIF - fuit
SUJET
GN
NOM - document
DET - le

PHRASE : le document fuit.

PHRASE
PREDICAT
VERB_TRANSITIF - annote
COMPLEMENT
GN
NOM - document
DET - le

SUJET
GN
NOM - service
DET - un

PHRASE : un service annote le document.

PHRASE
PREDICAT
VERB_TRANSITIF - réussit
COMPLEMENT
GN
NOM - document
DET - le
SUJET
GN
NOM - ministre
DET - le

PHRASE : le ministre réussit le document.

PHRASE
PREDICAT
VERB_TRANSITIF - annote
COMPLEMENT
GN
NOM - service
DET - un
SUJET
GN
NOM - document
DET - le

PHRASE : le document annote un service.

PHRASE
 PREDICAT
 VERB_TRANSITIF - réussit
 COMPLEMENT
 GN
 NOM - service
 DET - le
 SUJET
 GN
 NOM - service
 DET - le

PHRASE : le service réussit le service.

PHRASE
 PREDICAT
 VERB_INTRANSITIF - cherche
 SUJET
 GN
 NOM - coup
 DET - un

PHRASE : un coup cherche.

PHRASE
 PREDICAT
 VERB_INTRANSITIF - cherche
 SUJET
 GN

NOM - ministre
DET - un

PHRASE : un ministre cherche.

PHRASE
PREDICAT
VERB_INTRANSITIF - fuit
SUJET
GN
NOM - service
DET - un

PHRASE : un service fuit.

PHRASE
PREDICAT
VERB_TRANSITIF - réussit
COMPLEMENT
GN
NOM - service
DET - un
SUJET
GN
NOM - service
DET - le

PHRASE : le service réussit un service.

ANNEXE D

CATÉGORIES DU TREE-TAGGER

French TreeTagger Part-of-Speech Tags

Achim Stein, April 2003

ABR	abbreviation
ADJ	adjective
ADV	adverb
DET:ART	article
DET:POS	possessive pronoun (ma, ta, ...)
INT	interjection
KON	conjunction
NAM	proper name
NOM	noun
NUM	numeral
PRO	pronoun
PRO:DEM	demonstrative pronoun
PRO:IND	indefinite pronoun
PRO:PER	personal pronoun
PRO:POS	possessive pronoun (mien, tien, ...)
PRO:REL	relative pronoun
PRP	preposition
PRP:det	preposition plus article (au, du, aux, des)
PUN	punctuation
PUN:cit	punctuation citation
SENT	sentence tag
SYM	symbol
VER:cond	verb conditional
VER:futu	verb futur
VER:impe	verb imperative
VER:impl	verb imperfect
VER:infi	verb infinitive
VER:pper	verb past participle
VER:ppe	verb present participle
VER:pres	verb present
VER:simp	verb simple past
VER:subi	verb subjunctive imperfect
VER:subp	verb subjunctive present

BIBLIOGRAPHIE

(1992). Dictionnaire de l'Académie Française. Académie Française

Abeillé, Anne. 2007. *Les Grammaires d'Unification*. Paris: Hermès Sciences Publications.

----- (2008). Traitement Automatique des Langues. Encyclopædia Universalis En ligne. <<http://www.universalis-edu.com/encyclopedie/traitement-automatique-des-langues/#>>.

Bar-Hillel, Yehoshua. 1960. «The present Status of Automatic Translation of Languages». *Advances in Computers*, vol. 1, p. 91-141.

Bateman. 2002. «Natural Language Generation: an introduction and open-ended review of the state of the art». En ligne. <<http://www.fb10.uni-bremen.de/anglistik/langpro/webSPACE/jb/info-pages/nlg/ATG01/node1.html>>.

Beaune, Jean-Claude, André Doyon et Lucien Liaigre (2008). Automate. Encyclopædia Universalis En ligne. <<http://www.universalis-edu.com/encyclopedie/automate/>>.

Blache, Philippe. 2007. «Une introduction à HPSG». *2LC-CNRS*.

Bronckart, Jean-Paul. 1994. *Le fonctionnement du discours*. Lausanne-Paris: Delachaux & Niestlé S.A.

CNRTL. 2005. «Centre National de Ressources Textuelles et Lexicales». En ligne. <<http://www.cnrtl.fr/definition/>>.

- Cogswell Project. En ligne. <<http://www.cs.bham.ac.uk/research/projects/poplog/computers-and-thought/chap6/node5.html>>.
- CPAN. «CPAN». En ligne. <<http://search.cpan.org/>>.
- Cram, David, et Jaap Maat. 1998. «Dalgarno in Paris». *Histoire Épistémologie Langage*, p. 167-179. In *Persée* <http://www.persee.fr>. En ligne. <http://www.persee.fr/web/revues/home/prescript/article/hel_0750-8069_1998_num_20_2_2721>.
- Culioli, Antoine. 1999. *Pour une Linguistique de l'énonciation : Formalisation et opérations de repérages*, 2 t. Paris: Ophrys.
- Danlos, Laurence. 1985. *Génération Automatique de Textes en Langue Naturelle*: Masson.
- , 1990. «Génération automatique de textes en langue naturelle». *Texte et Ordinateur : Les mutations du Lire-Écrire*, vol. Hors Série, p. 18.
- Descartes, René. 1637. *Discours de la méthode - Partie V*.
- Dreyfus, H. L., et S. E. Dreyfus. 1986. *Mind over Machine: the power of human intuition and expertise in the era of the computer*. Oxford: Basil Blackwell.
- Duncan, Stewart (2009). Thomas Hobbes. Stanford Encyclopedia of Philosophy En ligne. <<http://plato.stanford.edu/entries/hobbes/>>.
- Faure, C., A. Grumbach, L. Likforman-Sulem et M. Sigelle (2005). Méthodes Structurelles et Neuronales En ligne. <http://formation.enst.fr/RDF/Docs/Poly_final_Mesen.pdf>.
- Friedman, Joyce. 1969. «A computer system for transformational grammar». *Communications of the ACM*, vol. 12, no 6, p. 8.
- Grevisse, Maurice. 2009. *Le petit Grevisse*, 32e: De Boeck - Duculot.
- Gruber, Thomas R. 1993. «A Translation Approach to Portable Ontology Specifications». *Knowledge Acquisition*, vol. 5, no 2, p. 199-220.
- Horridge, Matthew (2009). A practical guide to building OWL Ontology usinf Protégé 4 and Co-ODE Tools, University of Manchester
- Kahane, Sylvain (2002). Cours sur les grammaires formelles. Nanterre

- Kahane, Sylvain, et François Lareau. 2005. «Grammaire d'Unification Sens-Texte : modularité et polarisation». In *TALN*.
- Lepage, François. 2001. *Éléments de logique contemporaine*: Presses de l'Université de Montréal.
- Maisonneuve, Huguette. 2003. *Vade-Mecum de la nouvelle grammaire*, Version 2: Centre Collégial de Développement de Matériel Didactique.
- Miller, George A., et Christiane Fellbaum. 2007. «WordNet then and now». *Language Resources and Evaluation*, vol. 41, no 2, p. 209-214. In *Computer and Information Systems Abstracts*.
- Minsky, Marvin. 1975. «Minsky's frame system theory». In *Proceedings of the 1975 workshop on Theoretical issues in natural language processing*: Association for Computational Linguistics.
- Pollard, Carl, et Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*: The University of Chicago Press.
- Ponton, Claude (1997). GAT97, Université de Grenoble
- Rastier, François. 2004. «Ontologie(s)». *Revue des sciences et technologies de l'information*, vol. 18, no 1, p. 25.
- Robert, P. 1986. *Le petit Robert*. Paris: Le Robert.
- Sabah, Gérard. 1989. *L'intelligence artificielle et le langage - Processus de compréhension*. Paris: Hermes.
- Séris, Jean-Pierre. 1995. *Langages et machines à l'âge classique*. Paris: Hachette.
- Shieber, Stuart M., Fernando C. N. Pereira, Gertjan Van Noord et Robert C. Moore. 1990. «Semantic-Head-Driven Generation». *Computational Linguistics*, vol. 16, no 1.
- Sowa, J.F. 1983. *Conceptual structures: Information processing in mind and machine*.
- Stern, Jacques (2008). Cryptologie. Encyclopædia Universalis En ligne. <<http://www.universalis-edu.com/encyclopedie/cryptologie/>>.
- Swartout, William R. 1983. «XPLAIN: a system for creating and explaining expert consulting programs». *Artificial Intelligence*, vol. 21, no 3, p. 285-325.

- Tanguy, Ludovic, et Nabil Hathout. 2007. *Perl pour les linguistes : Programmes en Perl pour exploiter les données langagières*. Paris: Hermès Sciences Publications ; Lavoisier, 504 p.
- Tremblay, Ophélie. 2009. «Une ontologie des savoirs lexicologiques pour l'élaboration d'un module de cours en didactique du lexique». Montréal, Département de Didactique, Université de Montréal.
- Véronis, Jean. 2001. «INF Z18 : Informatique et Linguistique I». Université de Provence. En ligne. <<http://sites.univ-provence.fr/veronis/cours/INFZ18/>>.
- W3C. 2001. «Semantic Web». En ligne. <<http://www.w3.org/2001/sw/>>.
- Wall, Larry, Tom Christiansen et Jon Orwant. 2001. *Programmation en Perl*. Paris: O'Reilly, 1045 p.
- Weizenbaum, Joseph. 1976. *Computer Power and Human Reason*. San Francisco: Freeman.
- Wikipedia. «Flux RSS». En ligne. <[http://fr.wikipedia.org/wiki/RSS_\(format\)](http://fr.wikipedia.org/wiki/RSS_(format))>.
- Wilcock, Graham. 2007. «An OWL Ontology for HPSG». In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics: ACL 2007*.
- Woods, W. A. 1970. «Transition Network Grammar for Natural language Analysis». *Computational Linguistics*, vol. 13, no Number 10.
- Woolf, Beverly Park. 2009. *Building intelligent interactive tutors*: Morgan Kauffman.